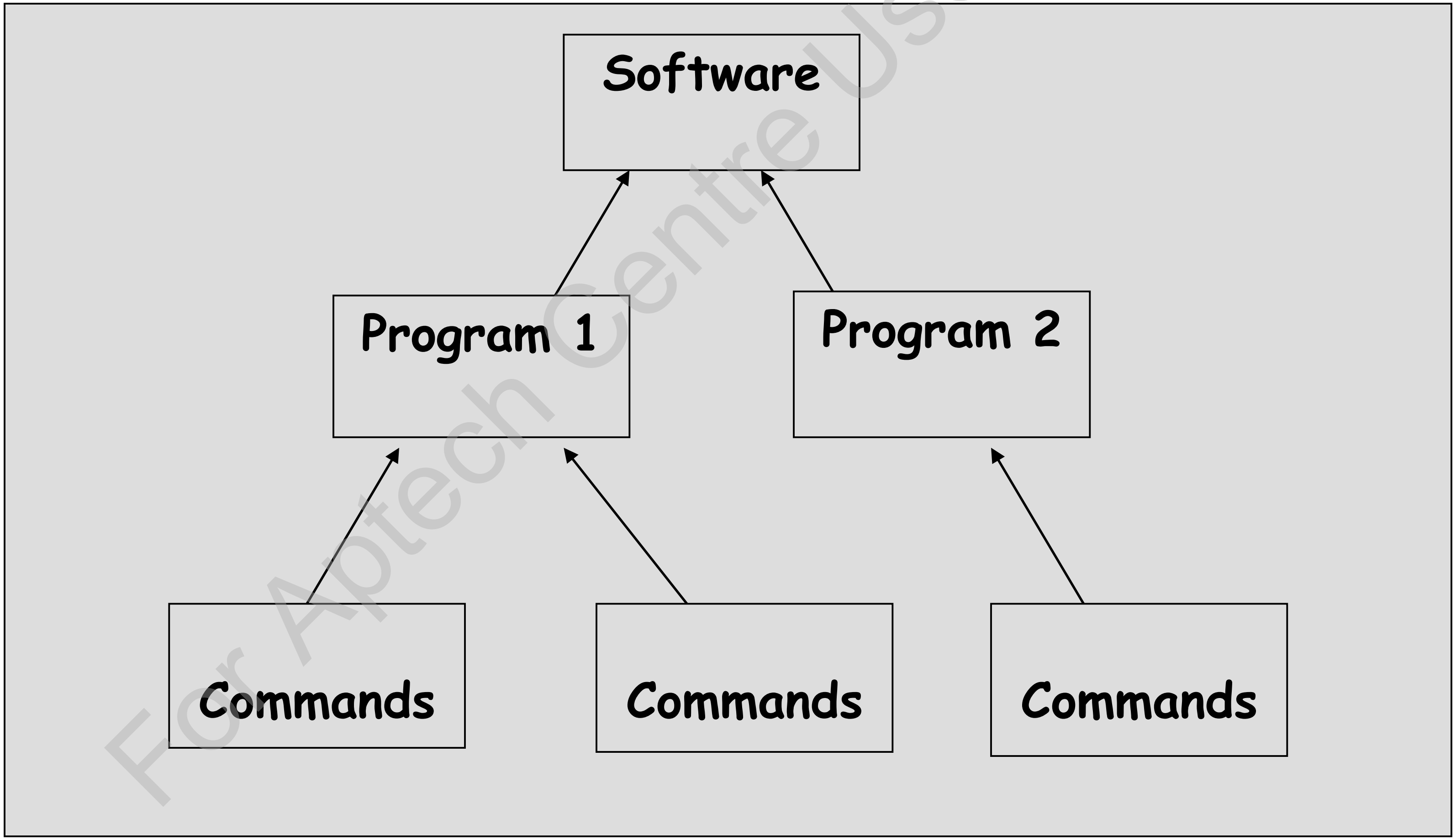
Basics of C

Session 1

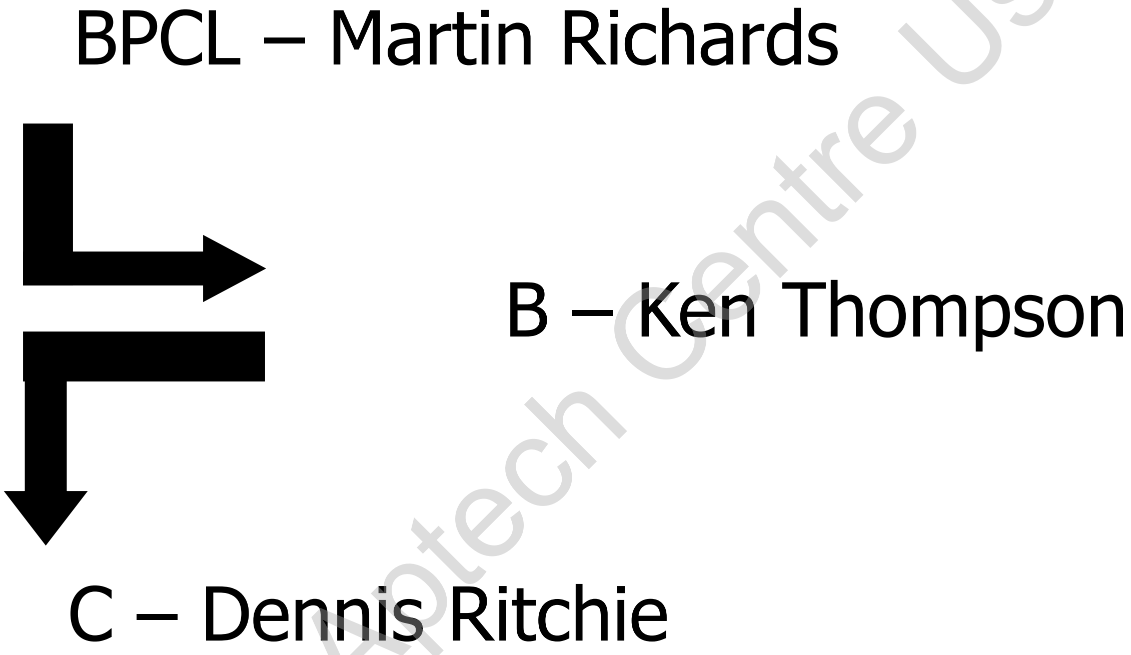
Objectives

- Differentiate between Command, Program and Software
- Explain the beginning of C
- Explain when and why is C used
- Discuss the C program structure
- Discuss algorithms
- Draw flowcharts
- List the symbols used in flowcharts

Software, Program and Command



The Beginning of C



BPCL – Martin Richards

B – Ken Thompson

C – Dennis Ritchie

Bell Laboratories, Inc.

Application Areas Of C

- C was initially used for systems programming
- A system program forms a portion of the operating system of the computer or its support utilities
- Operating Systems, Interpreters, Editors, Assembly programs are usually called system programs
- The UNIX operating system was developed using C
- There are C compilers available for almost all types of PC's

Middle Level Language

High Level Language

C

Assembly Language

Structured Language

- C allows compartmentalization of code and data

- It refers to the ability to sectic off and hide all information a instructions, necessary to perform a specific task, from rest of the program

```
do
{
         i = i + 1;
         .
         .
         .
} while (i < 40);
```

- Code can be compartmentalized in C by using functions or code blocks.

About C

- **C has 32 keywords**
- These keywords combined with a formal syntax form a C programming language
- Rules to be followed for all programs written in C:

◆ All keywords are lowercased

◆ C is case sensitive, do while is different from DO WHILE

◆ Keywords cannot be used as a variable or function name

```
main()
{
/* This is a sample Program*/
      int i,j;
      i=100;
      j=200;
        :
   }
```

The C Program Structure-1

main()

C programs are divided into units called functions.

Irrespective of the number of functions in a program, the operating system always passes control to main() when a C program is executed.

The function name is always followed by parentheses.

The parentheses may or may not contain parameters.

The C Program Structure-2

Delimiters { ... }

The function definition is followed by an open curly brace ({).

This curly brace signals the beginning of the function.

Similarly a closing curly brace (}) after the codes, in the function, indicate the end of the function.

The C Program Structure-3

Statement Terminator .... ;

A statement in C is terminated with a semicolon

A carriage return, whitespace, or a tab is not understood by the C compiler.

A statement that does not end in a semicolon is treated as an erroneous line of code in C.

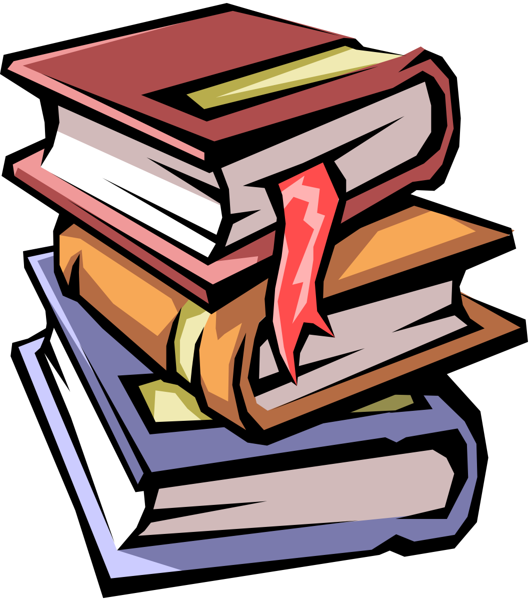
The C Program Structure-4

/* Comment Lines */

Comments are usually written to describe the task of a particular command, function or an entire program.

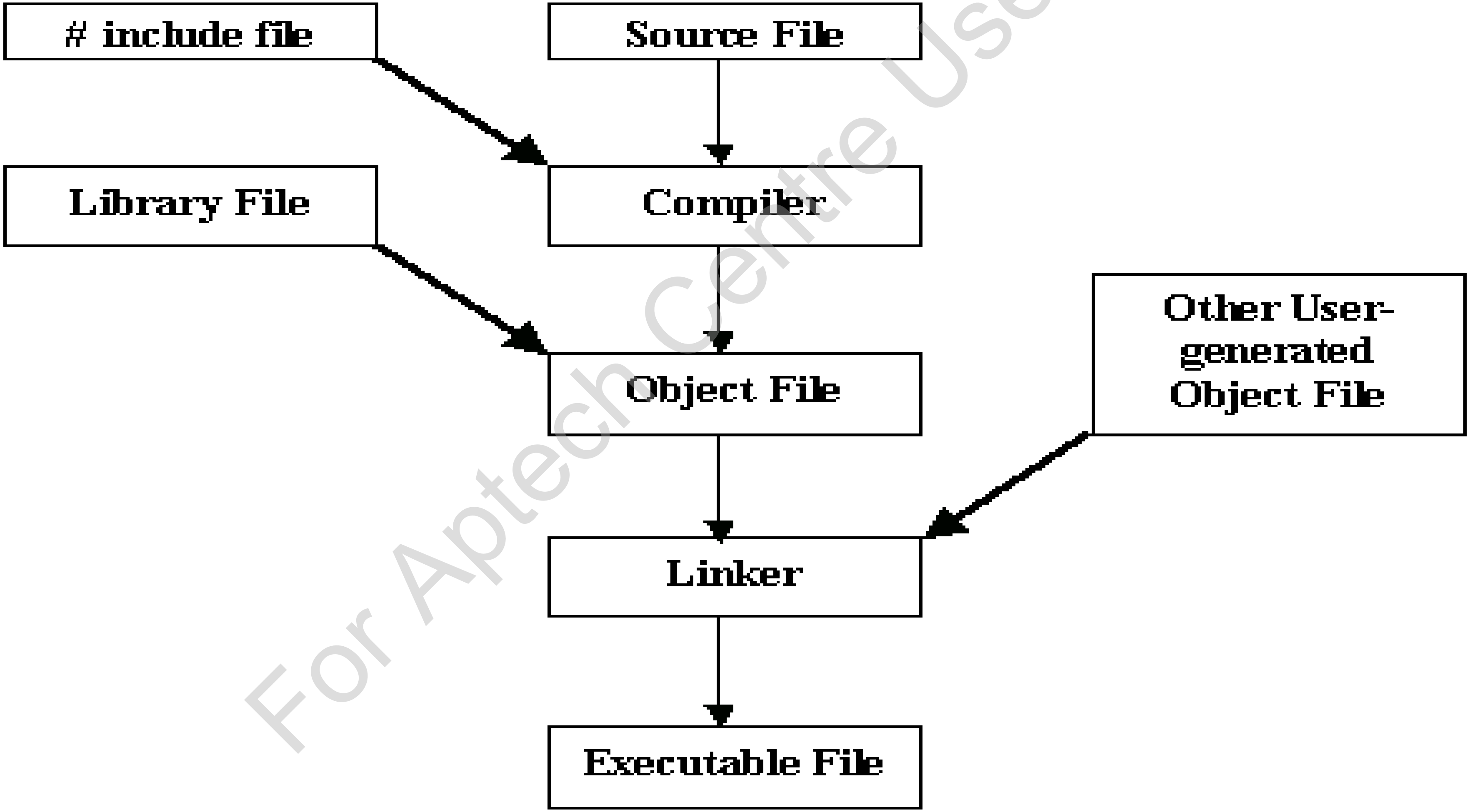
The compiler ignores them. In C, comments begin with /* and are terminated with */, in case the comments contain multiple lines

The C Library



- All C compilers come with a standard library of functions
- A function written by a programmer can be placed in the library and used when required
- Some compilers allow functions to be added in the standard library
- Some compilers require a separate library to be created

Compiling & Running A Program



include file

Source File

Library File

Compiler

Other User-generated Object File

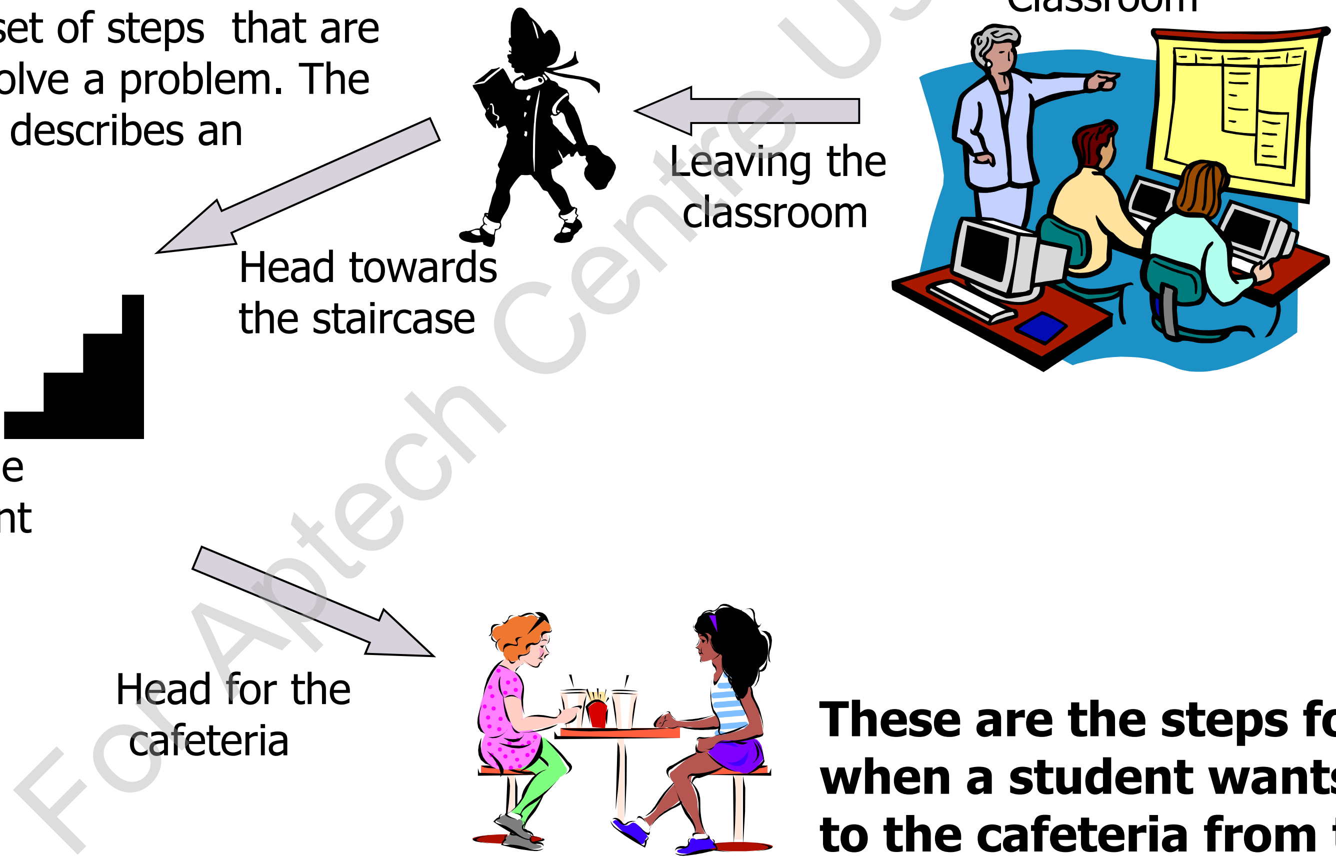
Object File

Linker

Executable File

The Programming Approach to Solving Problems

Algorithm is a set of steps that are performed to solve a problem. The example below describes an algorithm



Classroom

Leaving the classroom

Head towards the staircase

Go to the basement

Head for the cafeteria

Cafeteria

These are the steps followed when a student wants to go to the cafeteria from the classroom

Solving a Problem

In order to solve a problem

- Understand the problem clearly
- Gather the relevant information
- Process the information
- Arrive at the solution

Pseudocode

It is not actual code. A method of algorithm - writing which uses a standard set of words which makes it resemble code

```
BEGIN
DISPLAY 'Hello World !'
END
```

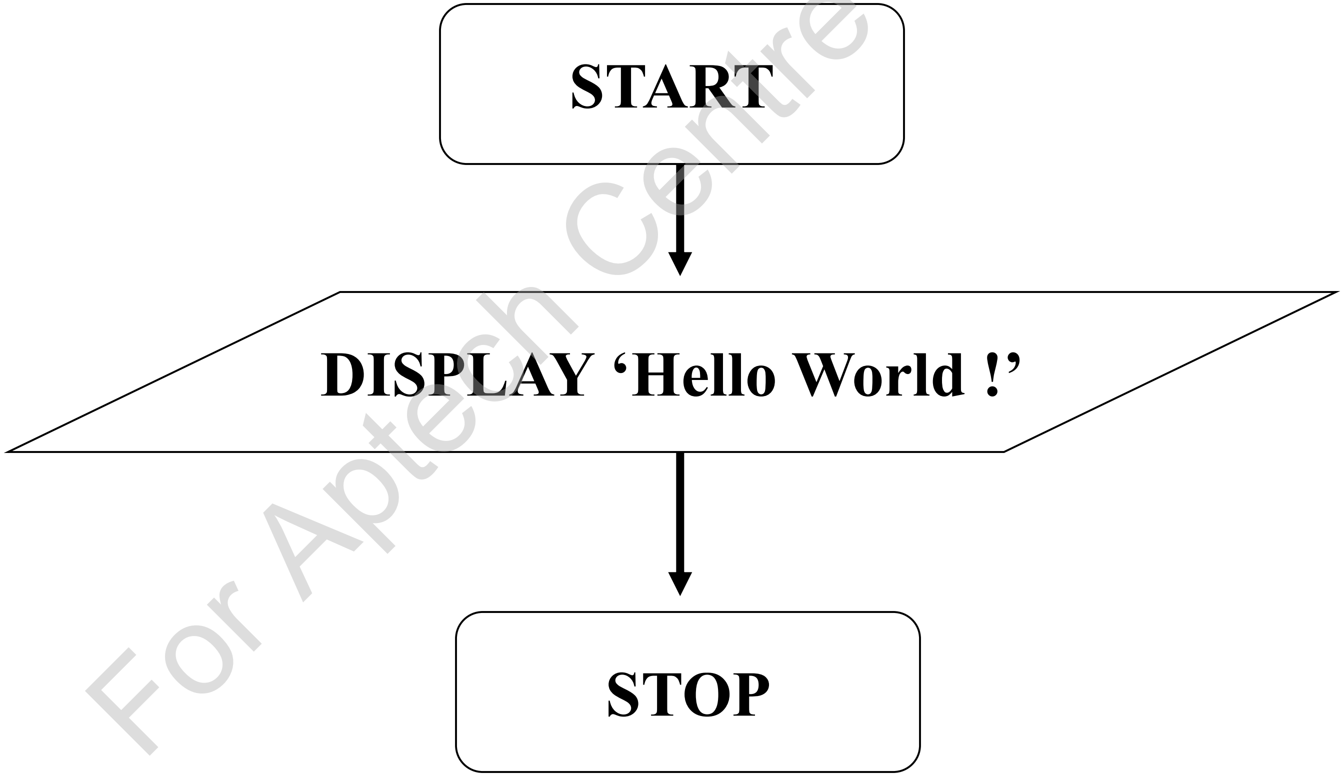
Each pseudocode starts with a BEGIN

To show some value , the word DISPLAY is used

The pseudocode finishes with an END

Flowcharts

It is a graphical representation of an algorithm

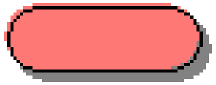
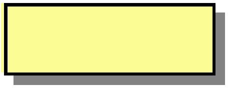
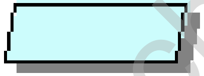
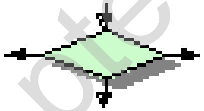
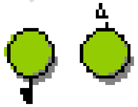
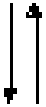


START

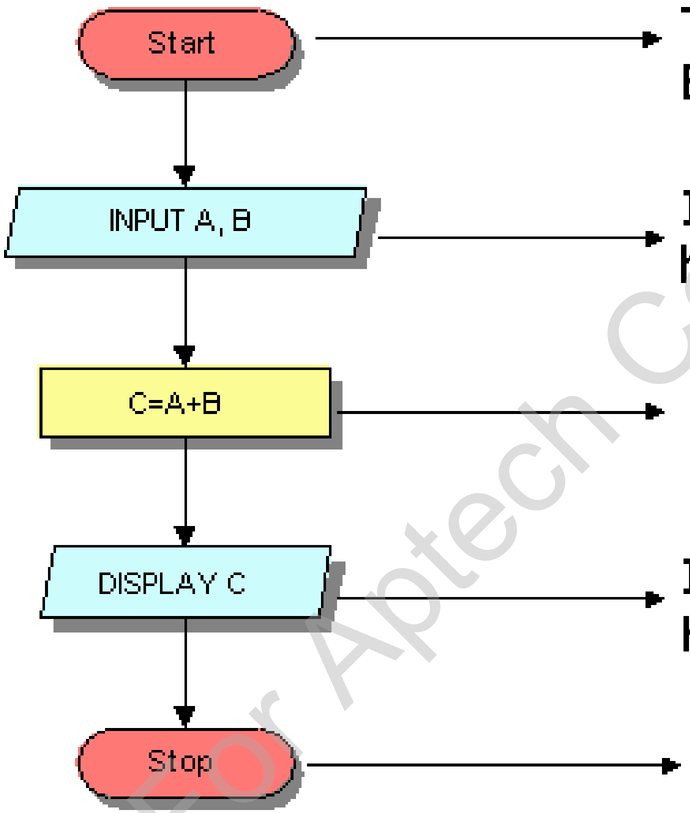
DISPLAY 'Hello World !'

STOP

The Flowchart Symbol

Symbol	Description
	Start or End of the Program
	Computational Steps
	Input / Output instructions
	Decision making & Branching
	Connectors
	Flow Line

Flowchart to add two numbers



Terminal Symbol: Used here to Begin the flowchart

Input/Output Symbol: Used here to Input two numbers

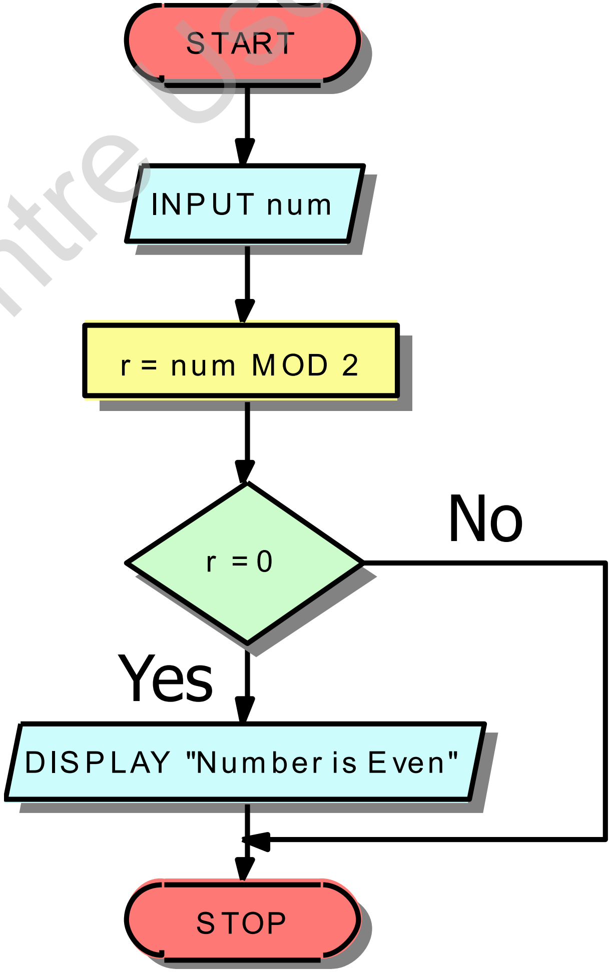
Process Symbol: Used here to add the two numbers

Input/Output Symbol: Used Here to Display the sum

Terminal Symbol: Used here to end the Flowchart

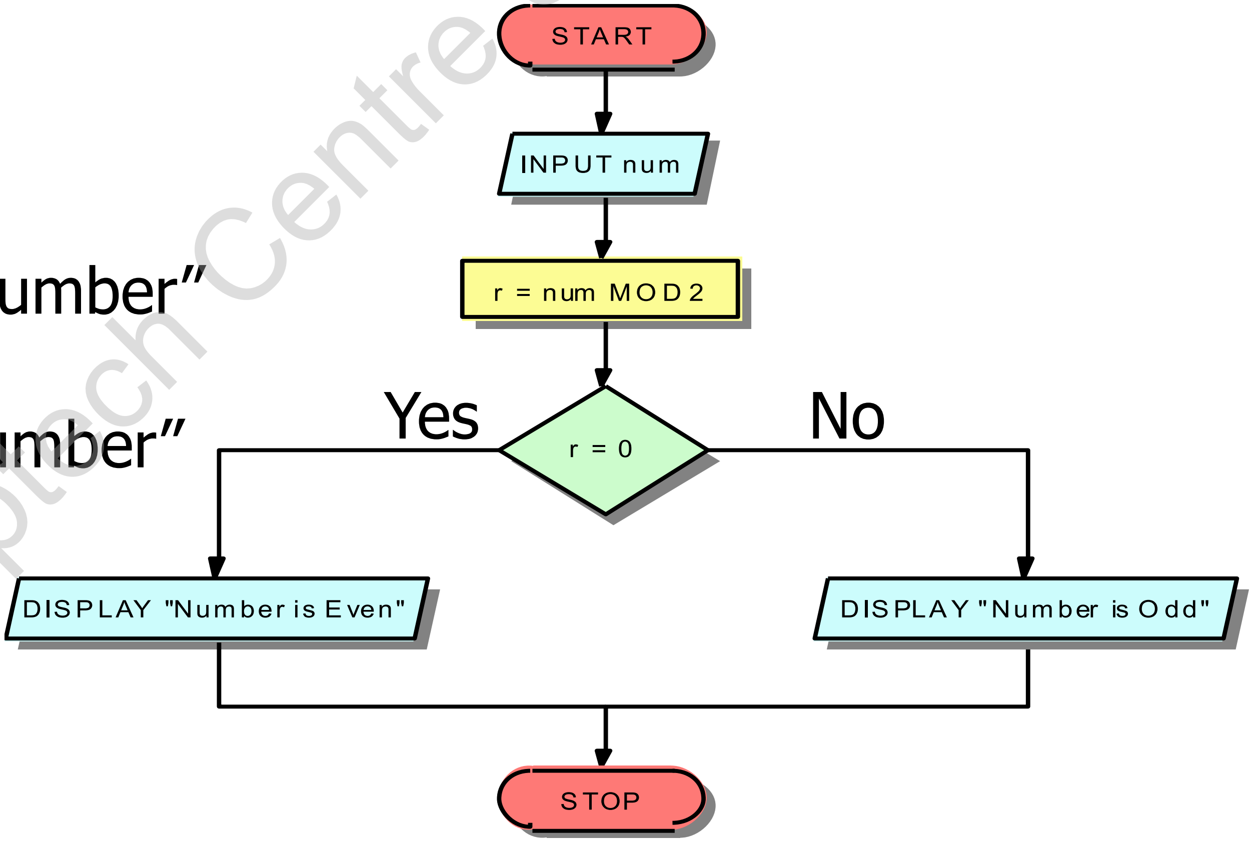
The IF Construct

```
BEGIN
INPUT num
r = num MOD 2
IF r=0
Display "Number is even"
END IF
END
```



The IF-ELSE Construct

```
BEGIN
INPUT num
r=num MOD 2
IF r=0
   DISPLAY "Even Number"
ELSE
   DISPLAY "Odd Number"
END IF
END
```



START

INPUT num

r = num MOD 2

r = 0

Yes

No

DISPLAY "Number is Even"

DISPLAY "Number is Odd"

STOP

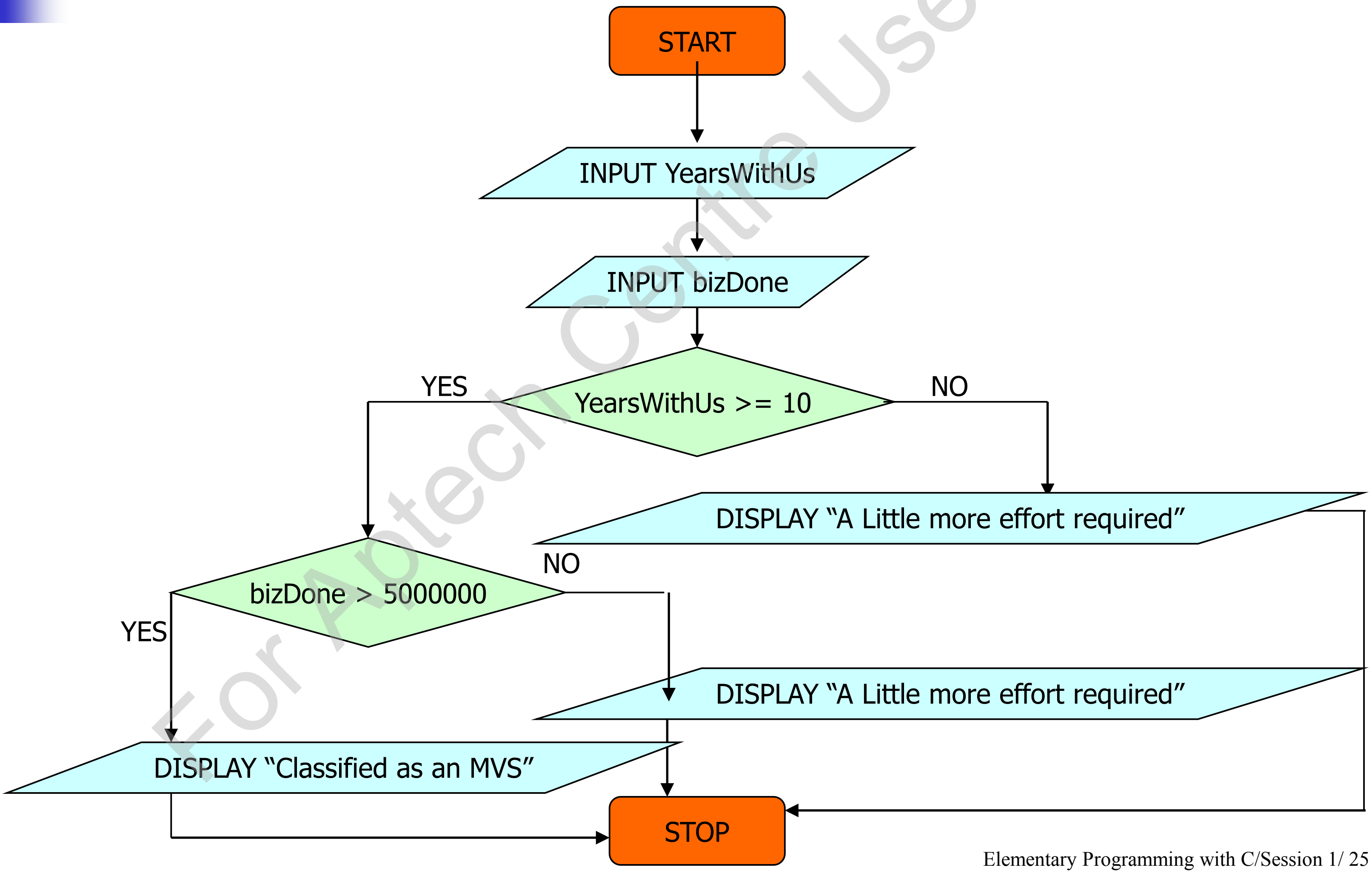
Multiple criteria using AND/OR

```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10 AND bizDone >=5000000
        DISPLAY “Classified as an MVS”
ELSE
        DISPLAY “A little more effort required!”
END IF
END
```

Nested IFs-1

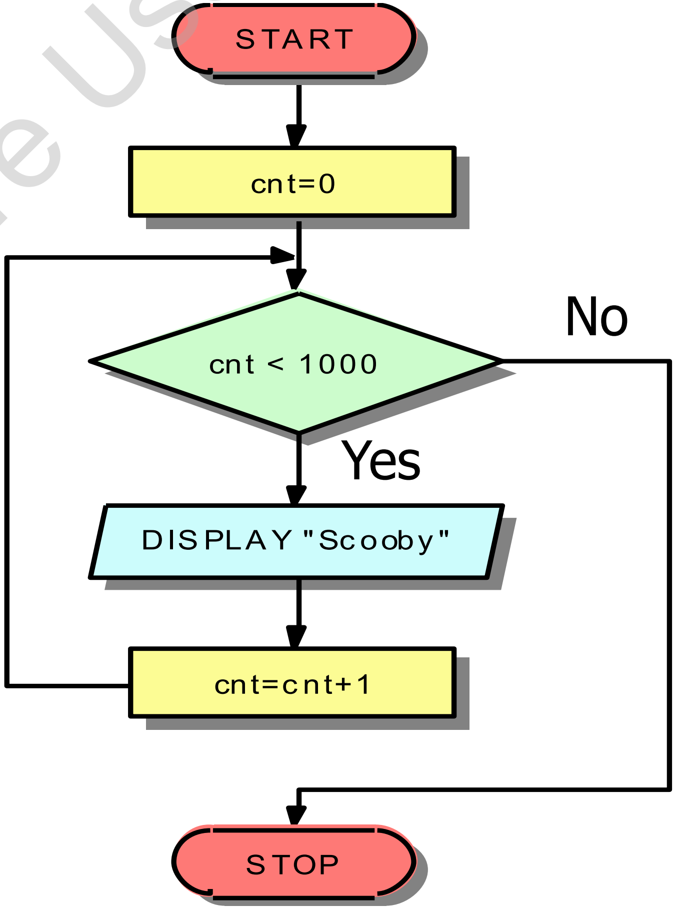
```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10
IF bizDone >=5000000
        DISPLAY "Classified as an MVS"
          ELSE
                DISPLAY "A little more effort required!"
END IF
ELSE
        DISPLAY "A little more effort required!"
END IF
END
```

Nested IFs-2


START
INPUT YearsWithUs
INPUT bizDone
YearsWithUs >= 10
YES
NO
DISPLAY "A Little more effort required"
bizDone > 5000000
NO
YES
DISPLAY "A Little more effort required"
DISPLAY "Classified as an MVS"
STOP

Loops

```
BEGIN
cnt=0
WHILE (cnt < 1000)
DO
        DISPLAY "Scooby"
        cnt=cnt+1
END DO
END
```



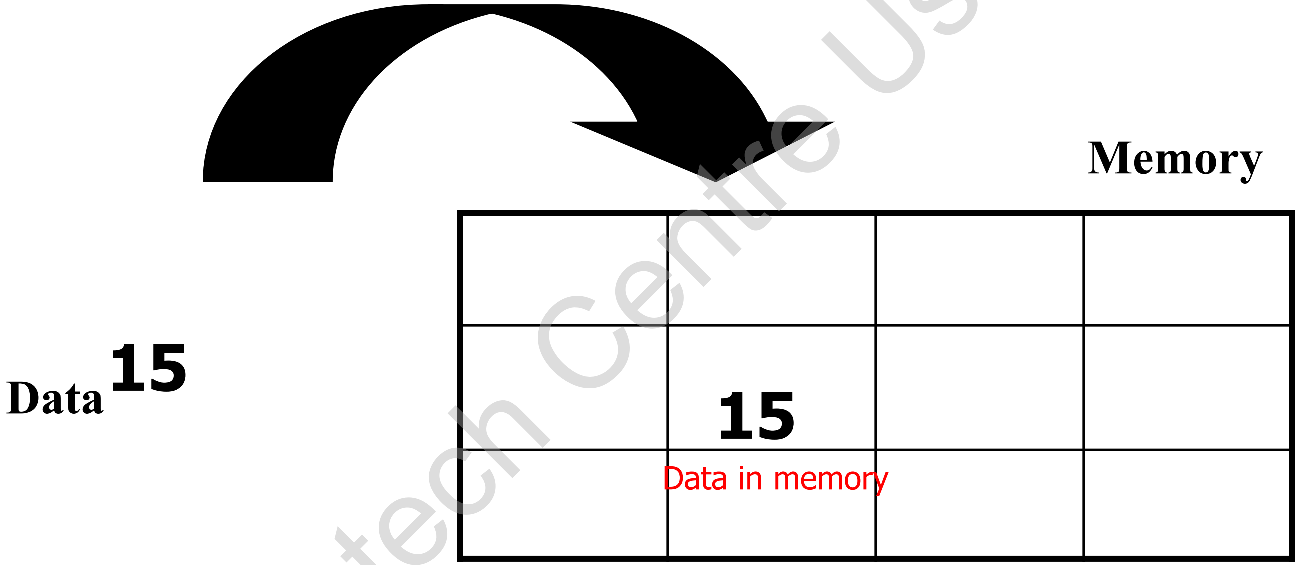
Variables and Data Types

Session 2

Objectives

- Discuss variables
- Differentiate between variables and constants
- List the different data types and make use of them in C programs
- Discuss arithmetic operators

Variables



Memory

Data 15

15

Data in memory

Each location in the memory is unique

Variables allow to provide a meaningful name for the location in memory

Example

```
BEGIN
 DISPlAY 'Enter 2  numbers'
INPUT A, B
C = A + B
DISPLAY  C
END
```

A, B and C are   variables in the pseudocode

Variable names takes away the need for a programmer to access memory locations using their address

The operating system takes care of allocating space for the variables

To refer to the value in the memory space, we need to only use the variable name

Constants

- A **constant** is a value whose worth never changes

- Examples

5	**numeric / integer constant**
5.3	**numeric / float constant**
'Black'	**string constant**
'C'	**Character constant**

- Variables hold constant values

Identifier Names

- The names of variables, functions, labels, and various other user defined objects are called identifiers
- Some correct identifier names
  - arena
  - s_count
  - marks40
  - class_one
- Examples of erroneous identifiers
  - 1sttest
  - oh!god
  - start... end

! is invalid

- Identifiers can be of any convenient length, but the number of characters in a variable that are recognized by a compiler varies from compiler to compiler
- Identifiers in C are case sensitive

Guidelines for Naming Identifiers

Variable names should begin with an alphabet

The first character can be followed by alphanumeric characters

Proper names should be avoided while naming variables

A variable name should be meaningful and descriptive

Confusing letters should be avoided

Some standard variable naming convention should be followed while programming

Keywords

- Keywords : All languages reserve certain words for their internal use
- Keywords hold a special meaning within the context of the particular language
- No problem of conflict as long as the keyword and the variable name can be distinguished. For example, having *integer* as a variable name is perfectly valid even though it contains the keyword int

Data Types-1

- Different types of data are stored in variables. Some examples are:
  - Numbers
    - Whole numbers. For example, 10 or 178993455
    - Real numbers. For example, 15.22 or 15463452.25
    - Positive numbers
    - Negative numbers
  - Names. For example, John
  - Logical values. For example, Y or N

Data Types-2

A data type describes the kind of data that will fit into a variable

The name of the variable is preceded with the data type

For example, the data type int would precede the name varName

```
Datatype variableName
```

```
int varName
```

Basic Data Types

The basic data types are

- `int`
- `float`
- `double`
- `char`
- `void`

Type int

- Stores numeric data

```
int num;
```

- Cannot then store any other type of data like "Alan" or "abc"
- 16 bits (2 bytes)
- Integers in the range -32768 to 32767
- Examples: 12322, 0, -232

Type float

- Stores values containing decimal places

```
float num;
```

- Precision of upto 6 digits
- 32 bits (4 bytes) of memory
- Examples: 23.05, 56.5, 32

Type double

- Stores values containing decimal places

```
double num;
```

- Precision of upto 10 digits
- 64 bits (8 bytes) of memory
- Examples: ‘a’, ‘m’, ‘$’ ‘%’ , ‘1’, ’5’

Type char

- Stores a single character of information

```
char gender;
gender='M';
```

- 8 bits (1 byte) of memory

- Examples: ‘a’, ‘m’, ‘$’ ‘%’ , ‘1’, ’5’

Type void

- Stores nothing
- Indicates the compiler that there is nothing to expect

Derived Data Types

Data type Modifiers + **Basic Data types** = **Derived data type**

unsigned + **int** = **unsigned int (Permits only positive numbers)**

short + **int** = **short int (Occupies less memory space than int)**

long + **int/double** = **Long int /longdouble (Occupies more space than int/double)**

signed and unsigned Types

- **unsigned** type specifies that a variable can take only positive values

```
unsigned int varNum;
varNum=23123;
```

- varNum is allocated 2 bytes
- modifier may be used with the **int** and **float** data types
- unsigned int supports range from 0 to 65535

long and short Types

- **short int** occupies 8 bits (1 byte)
  - allows numbers in the range -128 to 127
- **long int** occupies 32 bits (4 bytes)
  - 2,147,483,647 and 2,147,483,647
- **long double** occupies 128 bits (16 bytes)

Data Types and their range-1

Type	Approximate Size in Bits	Minimal Range
char	8	-128 to 127
unsigned	8	0 to 255
signed char	8	-128 to 127
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
signed int	16	Same as int
short int	16	Same as int
unsigned short int	8	0 to 65, 535

Data Types and their range-2

Type	Approximate Size in Bits	Minimal Range
signed short int	8	Same as short int
signed short int	8	Same as short int
long int	32	-2,147,483,647 to 2,147,483,647
signed long int	32	0 to 4,294,967,295
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	128	Ten digits of precision

Sample Declaration

```
main ()
{
   char abc;        /*abc of type character */
   int xyz;    /*xyz of type integer */
   float length;  /*length of type float */
   double area;  /*area of type double */
   long liteyrs;  /*liteyrs of type long int */
   short arm;      /*arm of type short integer*/
}
```

Operators and Expressions

Session 3

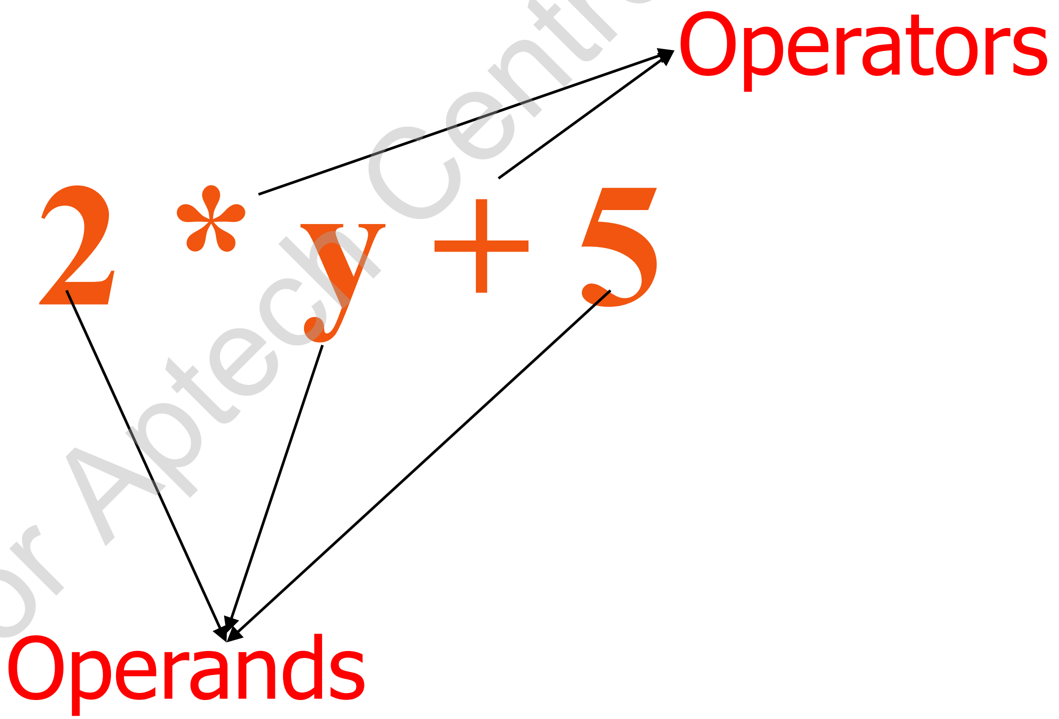
Objectives

- Explain Assignment Operator
- Understand Arithmetic Expressions
- Explain Relational and Logical Operators
- Understand Bitwise logical operators and expressions
- Explain casts
- Understand Precedence of Operators

Expressions

Combination of Operators and Operands

Example



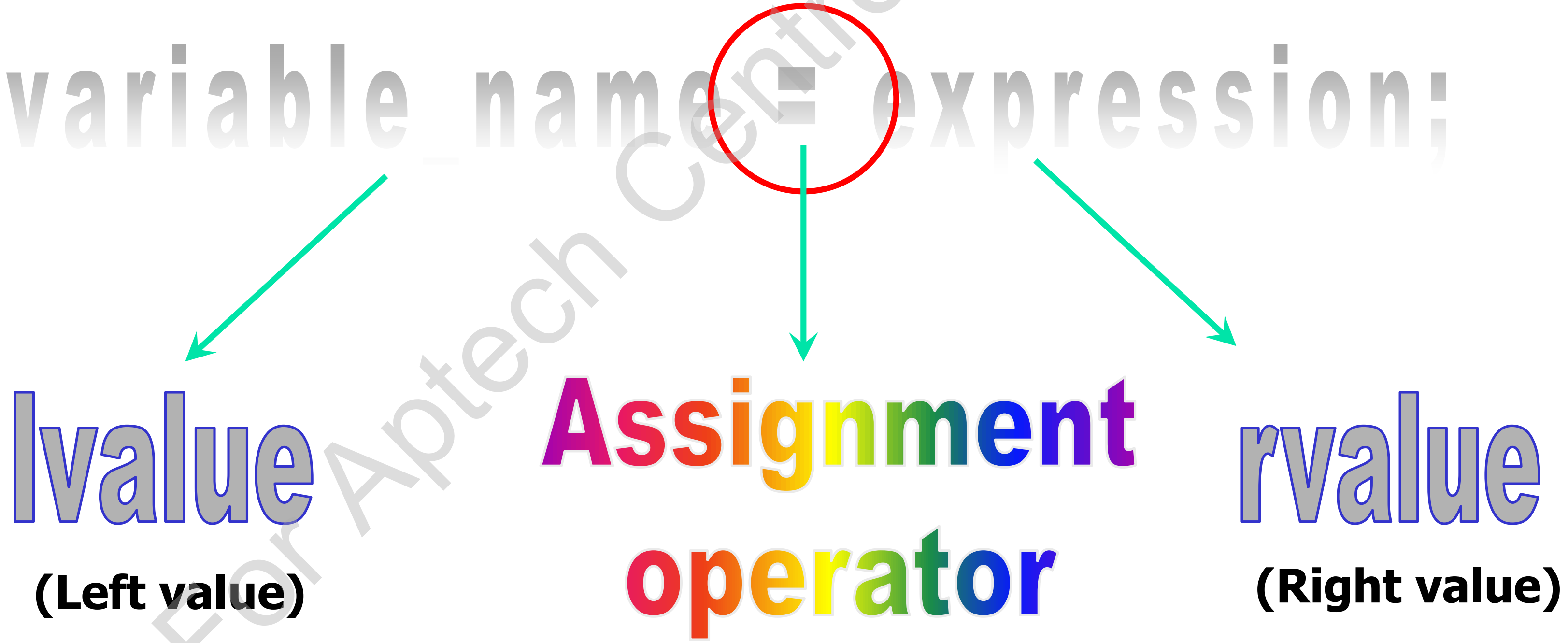
Operators

2 * y + 5

Operands

The Assignment Operator

The assignment operator(=) can be used with any valid C expression



variable_name = expression;

lvalue

(Left value)

Assignment operator

rvalue

(Right value)

Multiple Assignment

Many variables can be assigned the same value in a single statement

```
a = b = c = 10;
```

✓

However, you cannot do this :

```
int a = int b = int b = int c = 10
```

✗

Operators

4 Types

Arithmetic

Logical

Relational

Bitwise

Arithmetic Expressions

Mathematical expressions can be expressed in C using arithmetic operators

Examples

++i % 7

5 + (c = 3 + 8)

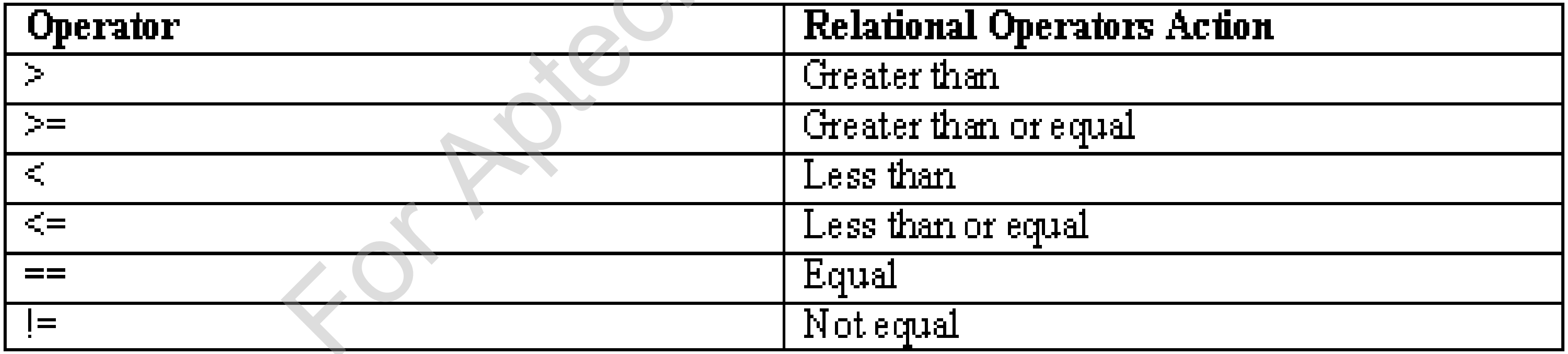
a * (b + c/d)22

Relational & Logical Operators-1

Used to……

Test the relationship between two variables, or between a variable and a constant

Relational Operators



Operator	Relational Operators Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

Relational & Logical Operators-2

Logical operators are symbols that are used to combine or negate expressions containing relational operators

Operator	Logical Operators Action
&&	AND
||	OR
!	NOT

Example: **if (a>10) && (a<20)**

Expressions that use logical operators return zero for false, and 1 for true

Bitwise Logical Operators-1

Processes data after converting number to its binary equivalent. (Bit wise representation)

AND (NUM1 & NUM2)	Return 1 if both the operands are 1
OR (NUM1 | NUM2)	Returns 1 if bits of either of the operand are 1
NOT (~ NUM1)	Reverses the bits of its operand (from 0 to 1 and 1 to 0)
XOR (NUM1 ^ NUM2)	Returns 1 if either of the bits in an operand is 1 but not both

Bitwise Logical Operators-2

Example

10 & 15→ 1010 & 1111→1010 → 10

10 | 15→ 1010 | 1111→1111 → 15

10 ^ 15→ 1010 ^ 1111→0101 → 5

~ 10 → ~1010 →1011 → -11

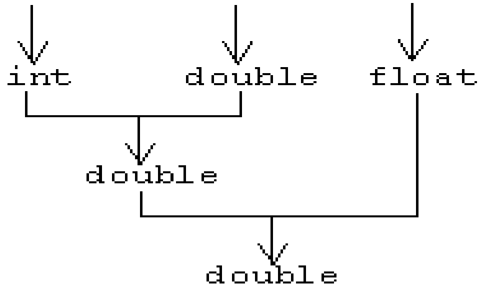
Type Conversion

The **automatic type conversions** for evaluating an expression are tabulated below.

a. **char** and **short** are converted to **int** and **float** is converted to **double.**
b. If either operand is **double**, the other is converted to **double**, and the result is **double**.
c. If either operand is **long**, the other is converted to **long** the result is **double.**
d. If either operand is **unsigned**, the other is also converted to **unsigned** and the result is also **unsigned**.
e. Otherwise all that are left are the operands of type **int,** and the result is **int.**

Example

```
char ch;
int i;
float f;
double d;
result = (ch/i) + (f*d) - (f+i);
```



Casts

An expression can be forced to be of a certain type by using a cast. The general syntax of cast:

(type) cast

type → any valid C data type

The integer value returned by (int)f
is converted back to floating point
when it crossed the assignment operator.
The value of f itself is not changed.

Example:

float x,f;

f = 3.14159;

x = (int) f; , the value of x will be 3 (integer)

Precedence Of Operators-1

- Precedence establishes the hierarchy of one set of operators over another when an arithmetic expression is to be evaluated
- It refers to the order in which C evaluates operators
- The precedence of the operators can be altered by enclosing the expressions in parentheses

Operator Class	Operators	Associativity
Unary	- ++ --	Right to Left
Binary	^	Left to Right
Binary	* / %	Left to Right
Binary	+ -	Left to Right
Binary	=	Right to Left

Precedence Of Operators-2

Example : -8 * 4 % 2 - 3

Sequence	Operation done	Result
1.	- 8 (unary minus)	negative of 8
2.	- 8 * 4	- 32
3.	- 32 % 2	16
4.	16-3	13

Precedence between comparison Operators

Always evaluated from left to right

Precedence for Logical Operators-1

Precedence	Operator
1	NOT
2	AND
3	OR

When multiple instances of a logical operator are used in a condition, they are evaluated from right to left

Precedence for Logical Operators-2

Consider the following expression

False OR True AND NOT False AND True

This condition gets evaluated as shown below:

False OR True AND [NOT False] AND True

NOT has the highest precedence.

False OR True AND [True AND True]

AND is the operator of the highest precedence and operators of the same precedence are evaluated from right to left

False OR [True AND True]

[False OR True]

True

Precedence among Operators-1

When an equation uses more than one type of operator then the order of precedence has to be established with the different types of operators

Precedence	Type of Operator
1	Arithmetic
2	Comparison
3	Logical

Precedence among Operators-2

Consider the following example:

2*3+4/2 > 3 AND 3<5 OR 10<9

The evaluation is as shown:

[2*3+4/2] > 3 AND 3<5 OR 10<9

First the arithmetic operators are dealt with

[[2*3]+[4/2]] > 3 AND 3<5 OR 10<9

[6+2] >3 AND 3<5 OR 10<9

[8 >3] AND [3<5] OR [10<9]

Precedence among Operators-3

Next to be evaluated are the comparison operators all of which have the same precedence and so are evaluated from left to right

True AND True OR False

The last to be evaluated are the logical operators. AND takes precedence over OR

[True AND True] OR False

True OR False

Changing Precedence-1

- Parenthesis ( ) has the highest level of precedence
- The precedence of operators can be modified using parenthesis ( )
- Operator of lower precedence with parenthesis assume highest precedence and gets executed first
- In case of nested Parenthesis ((( ))) the inner most parenthesis gets evaluated first
- An expression consisting of many set of parenthesis gets processed from left to right

Changing Precedence-2

Consider the following example:

5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (2<6 AND 10>11))

The solution is:

1. 5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (True AND False))

The inner parenthesis takes precedence over all other operators and the evaluation within this is as per the regular conventions

2. 5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR False)

Changing Precedence-3

3. 5+9*3^2-4 >10 AND (2+16-8/4 > 6 OR False)
   Next the outer parentheses is evaluated

4. 5+9*3^2-4 > 10 AND (2+16-2 > 6 OR False)
5. 5+9*3^2-4 > 10 AND (18-2 > 6 OR False)
6. 5+9*3^2-4 > 10 AND (16 > 6 OR False)
7. 5+9*3^2-4 > 10 AND (True OR False)
8. 5+9*3^2-4 > 10 AND True

Changing Precedence-4

9. 5+9*9-4>10 AND True
The expression to the left is evaluated as per the conventions
10. 5+81-4>10 AND True
11. 86-4>10 AND True
12. 82>10 AND True
13. True AND True
14. **True**

Input and Output in 'C'

Session 4

Objectives

- To understand formatted I/O functions - scanf() and printf()
- To use character I/O functions - getchar() and putchar()

Standard Input/Output

- In C, the standard library provides routines for input and output

- The standard library has functions for I/O that handle input, output, and character and string manipulation

- Standard input is usually the keyboard

- Standard output is usually the monitor (also called the console)

- Input and Output can be rerouted from or to files instead of the standard devices

The Header File <stdio.h>

- #include <stdio.h>
    - This is a preprocessor command
- stdio.h is a file and is called the header file
- contains the macros for many of the input/output functions used in 'C'
- printf(), scanf(), putchar(), getchar() functions are designed such that, they require the macros in stdio.h for proper execution

Formatted Input/Output

- printf() – for formatted output
- scanf() – for formatted input

- ***Format specifiers*** **specify the format in which the values of the variables are to be input and printed**

printf ()-1

- used to display data on the standard output – console

Syntax→ **printf ("control string", argument list);**

- The argument list consists of constants, variables, expressions or functions separated by commas
- There must be one format command in the control string for each argument in the list
- The format commands must match the argument list in number, type and order
- The control string must always be enclosed within double quotes, which are its delimiters

printf ()-2

The control string consists of one or more of three types of items:

1. **Text** characters:
   consists of printable characters
2. **Format Commands**:
   begins with a % sign and is followed by a format code - appropriate for the data item
3. **Nonprinting Characters:**
   Includes tabs, blanks and new lines

Format codes-1

Format	printf()	scanf()
Single Character	%c	%c
String	%s	%s
Signed decimal integer	%d	%d
Floating point (decimal notation)	%f	%f or %e
Floating point (decimal notation)	%lf	%lf
Floating point (exponential notation)	%e	%f or %e
Floating point (%f or %e , whichever is shorter)	%g	
Unsigned decimal integer	%u	%u
Unsigned hexadecimal integer (uses "ABCDEF")	%x	%x
Unsigned octal integer	%o	%o

In the above table c, d, f, lf, e, g, u, s, o and x are the type specifiers

Format codes-2

Format Code	Printing Conventions
%d	The number of digits in the integer
%f	The integer part of the number will be printed as such. The decimal part will consist of 6 digits. If the decimal part of the number is smaller than 6, it will be padded with trailing zeroes at the right, else it will be rounded at the right.
%e	One digit to the left of the decimal point and 6 places to the right , as in %f above

Control String Special Characters

\\	to print \ character
\ "	to print " character
%%	to print **%** character

control strings & format codes

No	Statements	Control String	What the control string contains	Argument List	Explanation of the argument list	Screen Display
1.	printf(“%d”,300);	%d	Consists of format command only	300	Constant	300
2.	printf(“%d”,10+5);	%d	Consists of format command only	10 + 5	Expression	15
3.	printf(“Good Morning Mr. Lee.”);	Good Morning Mr. Lee.	Consists of text characters only	Nil	Nil	Good Morning Mr. Lee.
4.	int count = 100; printf(“%d”,count);	%d	Consists of format command only	count	variable	100
5.	printf(“\nhello”);	\nhello	Consists of nonprinting character & text characters	Nil	Nil	hello on a new line
6.	#define str “Good Apple “ …….. printf(“%s”,str);	%s	Consists of format command only	Str	Symbolic constant	Good Apple

7.	……..
	int count,stud_num;
	count=0;
	stud_nim=100;
	printf(“%d %d\n”,count, stud_num);
		%d %d	Consists of format command and escape sequence	count, stud_num	two variables	0 , 100

Example for printf()

Program to display integer, float , character and string

```
#include <stdio.h>
   void main()
   {
                int a = 10;
                float b = 24.67892345;
                char ch = ‘A’;

                printf(“Integer data = %d”, a);
                printf(“Float Data = %f”,b);
                printf(“Character = %c”,ch);
                printf(“This prints the string”);
                printf(“%s”,”This also prints a string”);
    }
```

Modifiers in printf()-1

1. '-' Modifier

The data item will be *left-justified* within its field, the item will be printed beginning from the leftmost position of its field .

2. Field Width Modifier

Can be used with type float, double or char array (string). The field width modifier, which is an integer, defines , defines the *minimum* field width for the data item.

Modifiers in printf()-2

3. Precision Modifier

This modifier can be used with type float, double or char array (string). If used with data type float or double, the digit string indicates the *maximum* number of digits to be printed to the right of the decimal.

4. ‘0’ Modifier

The default padding in a field is done with spaces. If the user wishes to pad a field with zeroes this modifier must be used

5. ‘l’ Modifier

This modifier can be used to display integers as long int or a double precision argument. The corresponding format code is %ld

Modifiers in printf()-3

6. ‘h’ Modifier

This modifier is used to display short integers.
The corresponding format code is %hd

7. ‘*’ Modifier

If the user does not want to specify the field width in advance,
but wants the program to specify it, this modifier is used

Example for modifiers

```
/* This program demonstrate the use of Modifiers in printf() */
#include <stdio.h>
void main()
{
    printf(“The number 555 in various forms:\n”);
    printf(“Without any modifier: \n”);
    printf(“[%d]\n”,555);
    printf(“With - modifier :\n”);
    printf(“[%-d]\n”,555);
    printf(“With digit string 10 as modifier :\n”);
    printf(“[%10d]\n”,555);
    printf(“With 0 as modifier : \n”);
    printf(“[%0d]\n”,555);
    printf(“With 0 and digit string 10 as modifiers :\n”);
    printf(“[%010d]\n”,555);
    printf(“With -, 0 and digit string 10 as modifiers: \n”);
    printf(“[%-010d]\n”,555);
}
```

scanf()

- Is used to accept data

The general format of scanf() function

scanf(“control string”, argument list);

- The format used in the printf() statement are used with the same syntax in the scanf() statements too

Differences in argument list of between printf() and scanf()

- printf() uses variable names, constants , symbolic constants and expressions

- scanf() uses pointers to variables

When using scanf() follow these rules, for the argument list:

- If you wish to read in the value of a variable of basic data type, precede the variable name with a **&** symbol

- When reading in the value of a variable of derived data type, do not use a **&** before the variable name

Differences in the format commands of the printf() and scanf()

- There is no %g option
- The %f and %e format codes are in effect the same

Example for scanf()

```
#include <stdio.h>
  void main()
  {
      int a;
      float d;
      char ch, name[40];

   printf(“Please enter the data\n”);
   scanf(“%d %f %c %s”, &a, &d, &ch, name);

   printf(“\n The values accepted are :
      %d, %f, %c, %s”, a, d, ch, name);
  }
```

Buffered I/O

- used to read and write ASCII characters

A buffer is a temporary storage area, either in the memory, or on the controller card for the device

Buffered I/O can be further subdivided into:

- Console I/O
- Buffered File I/O

Console I/O

- Console I/O functions direct their operations to the standard input and output of the system

In ‘C’ the simplest console I/O functions are:

  - **getchar()** – which reads one (and only one) character from the keyboard

  - **putchar()** – which outputs a single character on the screen

getchar()

- Used to read input data, a character at a time from the keyboard
- Buffers characters until the user types a carriage return
- getchar() function has no argument, but the parentheses - must still be present

Example for getchar()

```
/* Program to demonstrate the use of getchar() */
#include <stdio.h>
void main()
{
  char letter;
  printf(“\nPlease enter any character : ‟);
  letter = getchar();
  printf(“\nThe character entered by you is %c‟, letter);
}
```

putchar()

- Character output function in ‘C’
- requires an argument

Argument of a putchar() function can be :

- A single character constant
- An escape sequence
- A character variable

putchar() options & effects

Argument	Function	Effect
character variable	putchar(c)	Displays the contents of character variable c
character constant	putchar(‘A’)	Displays the letter A
numeric constant	putchar(‘5’)	Displays the digit 5
escape sequence	putchar(‘\t’)	Inserts a tab space character at the cursor position
escape sequence	putchar(‘\n’)	Inserts a carriage return at the cursor position

putchar()

```
/* This program demonstrates the use of constants and escape
sequences in putchar() */

#include <stdio.h>
void main()
{
   putchar(‘H’); putchar(‘\n’);
   putchar(‘\t’);
   putchar(‘E’); putchar(‘\n’);
   putchar(‘\t’); putchar(‘\t’);
   putchar(‘L’); putchar(‘\n’);
   putchar(‘\t’); putchar(‘\t’); putchar(‘\t’);
   putchar(‘L’); putchar(‘\n’);
   putchar(‘\t’); putchar(‘\t’); putchar(‘\t’);
   putchar(‘\t’);
   putchar(‘O’);
}
```

Example

Condition

Session 5

Objectives

- Explain the Selection Construct
  - If Statement
  - If – else statement
  - Multi if statement
  - Nested if statement

- Switch statement

Conditional Statement

- Conditional statements enable us to change the flow of the program

- A conditional statement evaluates to either a true or a false value

Example :

To find whether a number is even or odd we proceed as follows :

1. **Accept a number**
2. **Find the remainder by dividing the number by 2**
3. **If the remainder is zero, the number is "EVEN"**
4. **Or if the remainder is not zero the number is "ODD"**

Selection Constructs

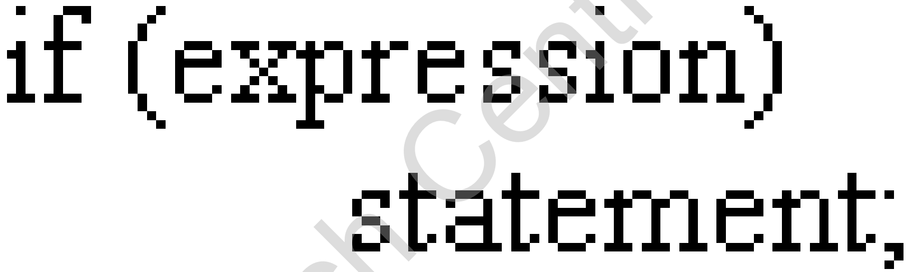
C supports two types of selection statements

The if statement

The switch statement

The if statement-1

Syntax:



```
if (expression)
        statement;
```

If the if expression evaluates to true, the block following the if statement or statements are executed

The if statement-2

Program to display the values based on a condition

```
#include <stdio.h>
void main()
{
   int x, y;
   char a = ‘y’;
   x = y = 0;
   if (a == ‘y’)
   {
      x += 5;
      printf(“The numbers are %d and \t%d”, x, y);
   }
}
```

Example

The if – else statement-1

Syntax:

```
if (expression)
            statement;
else
            statement;
```

The if – else statement-2

- If the if expression evaluates to true, the block following the if statement or statements are executed
- If the if expression does not evaluate to true then the statements following the else expression take over control
- The else statement is optional. It is used only if a statement or a sequence of statements are to be executed in case the if expression evaluates to false

The if – else statement -3

Program to display whether a number is Even or Odd

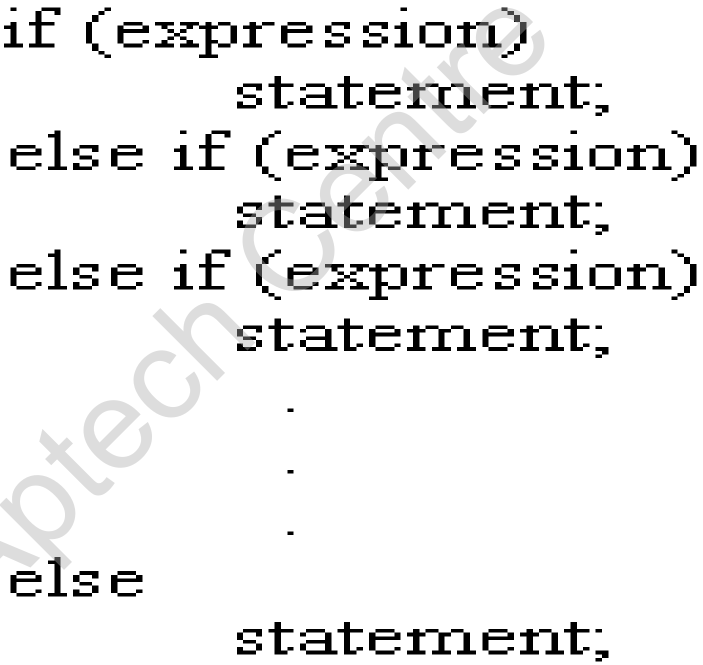
```
#include <stdio.h>
void main()
{
   int num , res ;

   printf(“Enter a number :”);
   scanf(“%d”,&num);
   res = num % 2;
   if (res == 0)
       printf(“Then number is Even”);
   else
       printf(“The number is Odd”);
}
```

Example

The if–else–if statement-1

Syntax:



```
if (expression)
          statement;
else if (expression)
          statement;
else if (expression)
          statement;
            .
            .
            .
else
          statement;
```

The if–else–if statement-2

- The if – else – if statement is also known as the if-else-if ladder or the if-else-if staircase

- The conditions are evaluated from the top downwards

The if–else–if statement-3

Program to display a message based on a value

```
#include <stdio.h>
   main()
   {
      int x;
      x = 0;
      clrscr ();
      printf(“Enter Choice (1 - 3) : “);
      scanf(“%d”, &x);
      if (x == 1)
         printf (“\nChoice is 1”);
      else if ( x == 2)
         printf (“\nChoice is 2”);
      else if ( x == 3)
         printf (“\nChoice is 3”);
      else
         printf (“\nInvalid Choice “);
   }
```

Example

Nested if-1

- The nested **if** is an **if** statement, which is placed within another **if** or **else**

- In C, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** statement and is not already associated with an **if**

Nested if-2

Syntax:

```
if (expl)
{
    if (exp2) statement1;
    if (exp3) statement2;
    else statement3;           /*with if (exp3) */
}
else statement4;              /* with if (expl) */
```

- Note that the inner else is associated with **if(exp3)**

- According to ANSI standards, a compiler should support at least 15 levels of nesting

Nested if-3

```
#include <stdio.h>
void main ()
 {
   int x, y;
   x = y = 0;
   clrscr ();
   printf (“Enter Choice (1 - 3) : “);
   scanf (“%d”, &x);
   if (x == 1)
   {
      printf(“\nEnter value for y (1 - 5) : “);
      scanf (“%d”, &y);
      if (y <= 5)
             printf(“\nThe value for y is : %d”, y);
      else
             printf(“\nThe value of y exceeds 5 “);
   }
   else
      printf (“\nChoice entered was not 1”);
}
```

Example

The switch statement-1

- The **switch** statement is a multi-way decision maker that tests the value of an expression against a list of integers or character constants

- When a match is found, the statements associated with that constant are executed

The switch statement-2

Syntax:

```
switch (expression)
{
        case constant1:
                statement sequence
                break;
        case constant2:
                statement sequence
                break;
        case constant3:
                statement sequence
                break;
        .
        .
        .
        default:
                statement sequence
}
```

The switch statement-3

Program to check whether the entered lowercase character is vowel or ‘z’ or a consonant

Example

```
#include <stdio.h>
  main ()
  {
     char ch;
     clrscr ();

     printf ("\nEnter a lower cased alphabet (a - z) : ");
     scanf("%c", &ch);
```

contd…….

The switch statement-4

```
if (ch < 'a' || ch > 'z')
        printf("\nCharacter not a lower cased alphabet");
else
        switch (ch)
        {
                case 'a' :
                case 'e' :
                case 'i' :
                case 'o' :
                case 'u' :
                        printf("\nCharacter is a vowel");
                        break;
                case 'z' :
                        printf ("\nLast Alphabet (z) was entered");
                        break;
                default :
                        printf("\nCharacter is a consonant");
                        break;
                        }
        }
```

Example

Loop

Session 6

Objectives

- Understand ‘for’ loop in ‘C’
- Work with comma operator
- Understand nested loops
- Understand the ‘while’ loop and the ‘do-while’ loop
- Work with break and continue statements
- Understand the exit() function

What is a Loop?

Section of code in a program which is executed repeatedly, until a specific condition is satisfied

3 types of Loop Structures

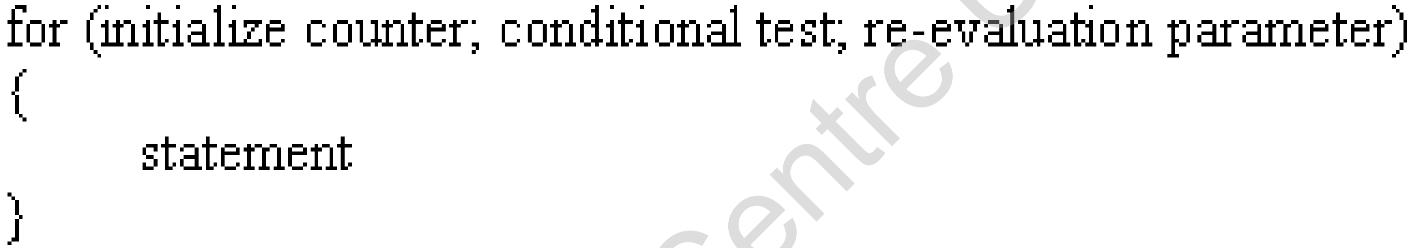
The for loop

The while loop

The do….while loop

The **for** loop-1

Syntax



```
for (initialize counter; conditional test; re-evaluation parameter)
{
        statement
}
```

- The initialize counter is an assignment statement that sets the loop control variable, before entering the loop
- The conditional test is a relational expression, which determines, when the loop will exit
- The evaluation parameter defines how the loop control variable changes, each time the loop is executed

The **for** loop-2

- The three sections of the **for** loop must be separated by a semicolon(;)

- The statement, which forms the body of the loop, can either be a single statement or a compound statement

- The **for** loop continues to execute as long as the conditional test evaluates to true. When the condition becomes false, the program resumes on the statement following the **for** loop

The for loop-3

```
/*This program demonstrates the for loop in a C program */
  #include <stdio.h>

  main()
  {
            int count;
            printf(“\tThis is a \n”);

            for(count = 1; count <=6 ; count++)
                 printf(“\n\t\t nice”);

            printf(“\n\t\t world. \n”);
  }
```

Example

The Comma Operator

The scope of the **for** loop can be extended by including more than one initializations or increment expressions in the for loop specification

The format is : **exprn1 , exprn2 ;**

Example

```
#include <stdio.h>
main()
{
            int i, j , max;
            printf(“Please enter the maximum value \n”);
            printf(“for which a table can be printed: “);
            scanf(“%d”, &max);

            for(i = 0 , j = max ; i <=max ; i++, j--)
                  printf(”\n%d  +  %d  =  %d”,i, j, i + j);
}
```

Nested **for** Loops-1

The **for** loop will be termed as a **nested for** loop when it is written as follows

```
for(i = 1; i<max1; i++)
        {
                                .
                                .
                for(j = 0; j < = max2; j++)
                {
                        .
                        .
                }
                .
                .
        }
```

Nested **for** Loops-2

```
#include <stdio.h>
  main()
  {
    int i, j, k;
    i = 0;
    printf("Enter no. of rows :");
    scanf("%d", &i);
    printf("\n");
    for (j = 0; j < i ; j++)
    {
      printf("\n");
      for (k = 0; k <= j; k++) /*inner for loop*/
      printf("*");
    }
  }
```

Example

The **while** Loop-1

Syntax

```
while (condition is true)
  statement ;
```

The while loop repeats statements while a certain specified condition is True

The **while** Loop-2

```
/* A simple program using the while loop */

  #include <stdio.h>
  main()
     {
      int count = 1;
      while( count <= 10)
      {
            printf(“\n This is iteration %d\n”,count);
            count++;
      }

      printf(“\n The loop is completed. \n”);
         }
```

Example

do...while Loop-1

Syntax

```
do{
        statement;
     } while (condition);
```

- In the do while loop the body of the code is executed once before the test is performed
- When the condition becomes False in a **do while** the loop will be terminated, and the control goes to the statement that appears immediately after the while statement

do...while Loop-2

Example

```
#include <stdio.h>
   main ()
    {
   int num1, num2;
             num2 = 0;
   do
   {
        printf( "\nEnter a number : ");
        scanf(“%d”,&num1);
        printf( " No. is %d",num1);
        num2++;
   } while (num1 != 0);
   printf ("\nThe total numbers entered were %d",--num2);

/*num2 is decremented before printing because count for last
integer (0) is not to be considered */
   }
```

Jump Statements-1

return expression

- The return statement is used to return from a function
- It causes execution to return to the point at which the call to the function was made
- The return statement can have a value with it, which it returns to the program

Jump Statements-2

goto label

- The goto statement transfers control to any other statement within the same function in a C program
- It actually violates the rules of a strictly structured programming language
- They reduce program reliability and make program difficult to maintain

Jump Statements-3

break statement

- The break statement is used to terminate a case in a switch statement
- It can also be used for abrupt termination of a loop
- When the break statement is encountered in a loop, the loop is terminated immediately and control is passed to the statement following the loop

break statement

Example

```
#include <stdio.h>
  main ()
  {
     int count1, count2;
     for(count1 = 1, count2 = 0;count1 <=100; count1++)
     {
        printf("Enter %d count2 : ",count1);
        scanf("%d", &count2);
        if(j==100) break;
     }
```

Jump Statements-4

continue statement

- The continue statement causes the next iteration of the enclosing loop to begin
- When this statement is encountered, the remaining statements in the body of the loop are skipped and the control is passed on to the re-initialization step

continue statement

Example

```
#include <stdio.h>
  main ()
  {
    int num;
    for(num = 1; num <=100; num++)
    {
      if(num % 9 == 0)
           continue;
      printf("%d\t",num);
    }
  }
```

Jump Statements-5

exit() function

- The exit() is used to break out of the program

- The use of this function causes immediate termination of the program and control rests in the hands of the operating system

Arrays

Session 7

Objectives

- Explain array elements and indices
- Define an array
- Explain array handling in C
- Explain how an array is initialized
- Explain string / character arrays
- Explain two dimensional arrays
- Explain initialization of two dimensional arrays

Array Elements & Indices

- Each member of an array is identified by unique index or subscript assigned to it
- The dimension of an array is determined by the number of indices needed to uniquely identify each element
- An index is a positive integer enclosed in [ ] placed immediately after the array name
- An index holds integer values starting with zero
- An array with 11 elements will look like -

Player[0], player[1], player[2],…. Player[10]

Defining an Array-1

- An array has some particular characteristics and has to be defined with them
- These characteristics include –

Storage Class

Data Types of the elements in the Array

Array Name Which indicates the location of the first member of the array

Array Size a constant evaluating to a +ve value

Defining an Array-2

An array is defined in the same way as a variable is defined. The only change is that the array name is followed by one or more expressions, enclosed within square brackets [], specifying the array dimension.

```
Storage_Class   data_types   array_name[size]
          int player[11];
```

Norms with Arrays

- All elements of an array are of the same type
- Each element of an array can be used wherever a variable is allowed or required
- Each element of an array can be referenced using a variable or an integer expression
- Arrays can have their data types like int, char, float or double

Array Handling in C-1

- An array is treated differently from a variable in C
- Two arrays, even if they are of the same type and size cannot be tested for equality
- It is not possible to assign one array directly to another
- Values cannot be assigned to an array on the whole, instead values are assigned to the elements of the array

Array Handling in C-2

```
/* Input values are accepted from the user into the array ary[10]*/
#include <stdio.h>
void main()
{
    int ary[10];
    int i, total, high;
    for(i=0; i<10; i++)
    {
    printf(“\n Enter value: %d : ”, i+1);
    scanf(“%d”,&ary[i]);
     }
     /* Displays highest of the entered values */
     high = ary[0];
     for(i=1; i<10; i++)
     {
    if(ary[i] > high)
       high = ary[i];
     }
     printf(“\nHighest value entered was %d”, high);
     /* prints average of values entered for ary[10] */
     for(i=0,total=0; i<10; i++)
    total = total + ary[i];
     printf(“\nThe average of the elements of ary is %d”,total/i);
}
```

Array Initialization

- Each element of an Automatic array needs to be initialized separately
- In the following example the array elements have been assigned valued using the for loop

```
      #include <stdio.h>
void main()
{
   char alpha[26];
   int i, j;
   for(i=65,j=0; i<91; i++,j++)
   {
   alpha[j] = i;
   printf("The character now assigned is %c \n", alpha[j]);
   }
   getchar();
}
```

- In case of extern and static arrays, the elements are automatically initialized to zero

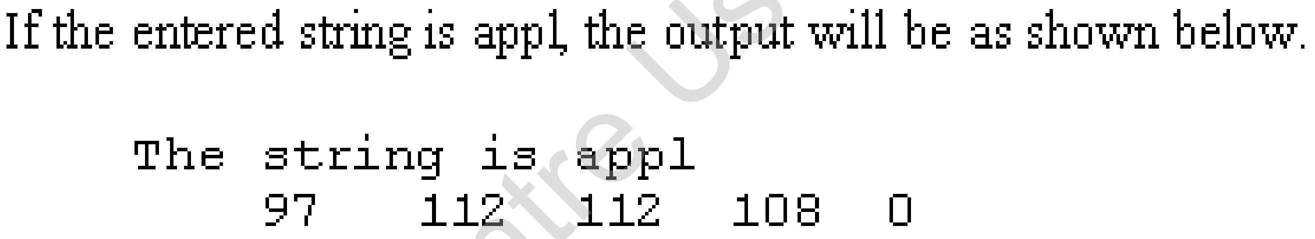
Strings/Character Arrays-1

- A string can be defined as a character type array, which is terminated by a null character
- Each character in a string occupies one byte and the last character of a string is “\0” (Backslash zero)
- Example

```
#include <stdio.h>
void main()
{
   char ary[5];
   int i;
   printf(“\n Enter string : “);
   scanf(“%s”,ary);
   printf(“\n The string is %s \n\n”, ary);
   for (i=0; i<5; i++)
   printf(“\t%d”, ary[i]);
 }
```

Strings/Character Arrays-2

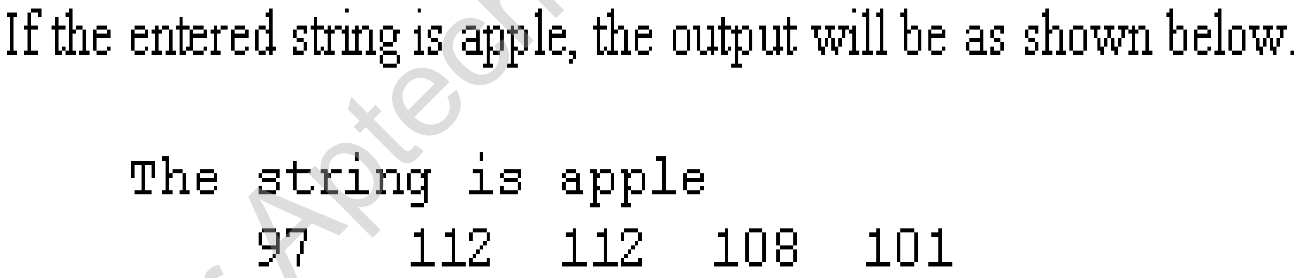
Output -



If the entered string is appl, the output will be as shown below.

```
The string is appl
    97   112  112  108  0
```

The input for the above is of 4 characters and the 5th character is the null character



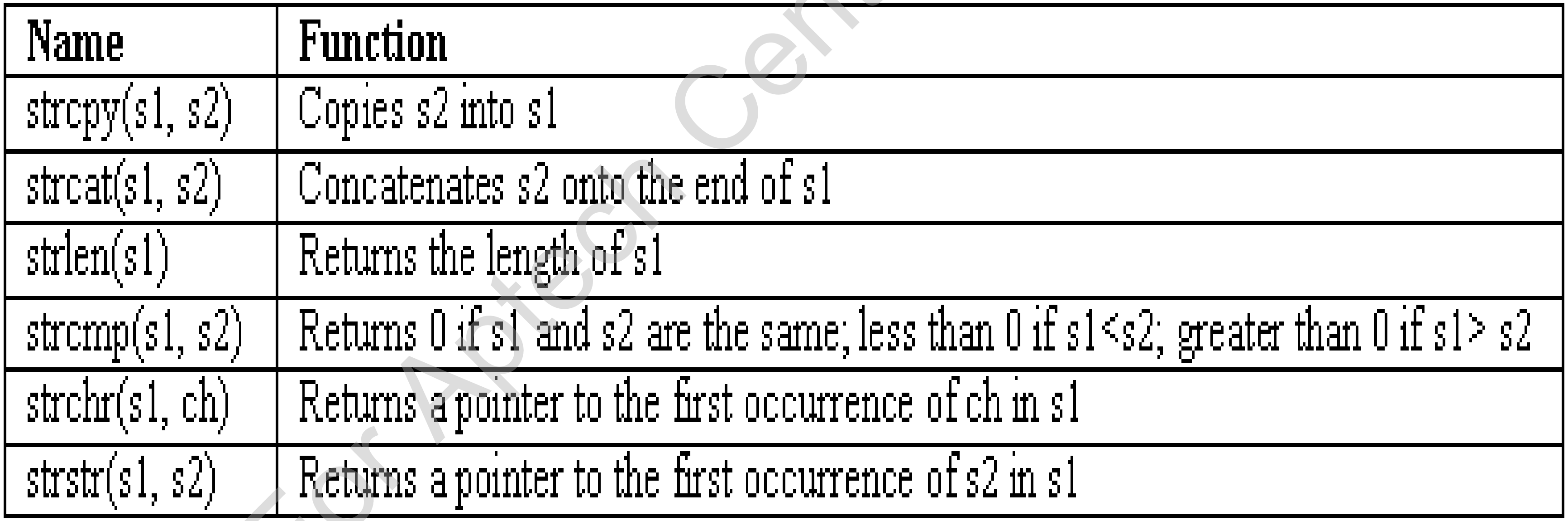
If the entered string is apple, the output will be as shown below.

```
The string is apple
    97   112  112  108  101
```

The above output is for an input of 5 characters

String Functions

Wide range of string functions, which are found in the standard header file <string.h>



Name	Function
strcpy(s1, s2)	Copies s2 into s1
strcat(s1, s2)	Concatenates s2 onto the end of s1
strlen(s1)	Returns the length of s1
strcmp(s1, s2)	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1> s2
strchr(s1, ch)	Returns a pointer to the first occurrence of ch in s1
strstr(s1, s2)	Returns a pointer to the first occurrence of s2 in s1

Two-Dimensional Arrays

- The simplest and the most commonly used multi-dimensional array is the two - dimensional array
- A two-dimensional array can be thought of as an array of two single dimensional arrays
- A two-dimensional array looks like a railway time-table consisting of rows and columns
- A two–dimensional array is declared as -

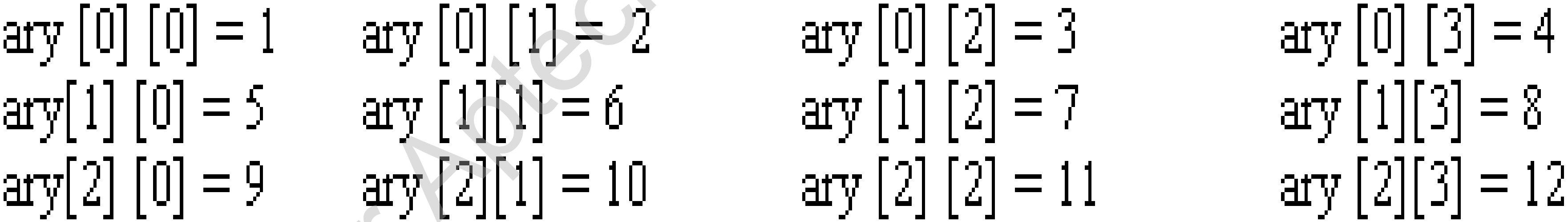
int temp[4][3];

Initialization of Multidimensional Arrays-1

```
int ary[3][4] =

   {1,2,3,4,5,6,7,8,9,10,11,12};
```

The result of the above assignment will be as follows :



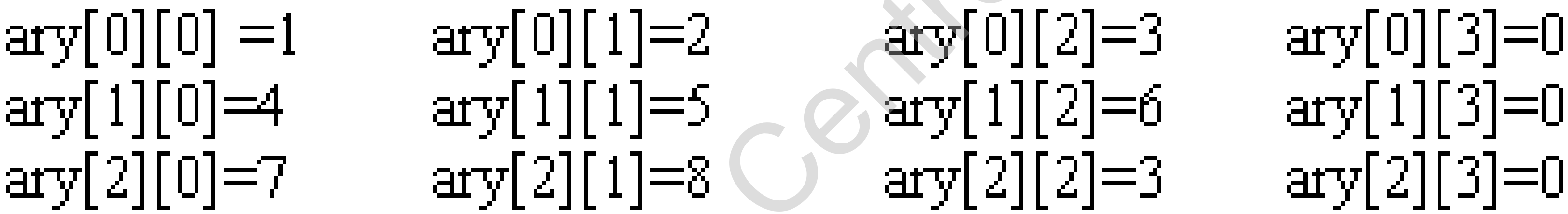
ary [0] [0] = 1   ary [0] [1] = 2   ary [0] [2] = 3   ary [0] [3] = 4
ary[1] [0] = 5   ary [1][1] = 6   ary [1] [2] = 7   ary [1][3] = 8
ary[2] [0] = 9   ary [2][1] = 10   ary [2] [2] = 11   ary [2][3] = 12

Initialization of Multidimensional Arrays-2

```
int ary[3][4]=
                  {
                         {1,2,3},
                         {4,5,6},
                         {7,8,3}
              };
```

Initialization of Multidimensional Arrays-3

The result of the assignment will be as follows :



```
ary[0][0] =1     ary[0][1]=2     ary[0][2]=3     ary[0][3]=0
ary[1][0]=4      ary[1][1]=5     ary[1][2]=6     ary[1][3]=0
ary[2][0]=7      ary[2][1]=8     ary[2][2]=3     ary[2][3]=0
```

A two - dimensional string array is declared in the following manner :

char str_ary[25][80];

Two-Dimensional Array-1

Example

```
#include <stdio.h>
#include <string.h>
void main ()
{
   int i, n = 0;
   int item;
   char x[10][12];
   char temp[12];

   clrscr();
   printf(“Enter each string on a separate line\n\n”);
   printf(“Type ‘END’ when over \n\n”);

   /* read in the list of strings */
   do
   {
      printf(“String %d : ”, n+1);
      scanf(“%s”, x[n]);
   } while (strcmp(x[n++], “END”));
/*reorder the list of strings */
```

contd….

Two-Dimensional Array-2

Example

```
n = n - 1;
    for(item=0; item<n-1; ++item)
    {
        /* find lowest of remaining strings */
            for(i=item+1; i<n; ++i)
            {
                        if(strcmp (x[item], x[i]) > 0)
                        {
                                   /*interchange two stings */
                                   strcpy (temp, x[item]);
                                   strcpy (x[item], x[i]);
                                   strcpy (x[i], temp);
                        }
              }
       }
/* Display the arranged list of strings */
printf("Recorded list of strings : \n");
      for(i = 0; i < n ; ++i)
      {
           printf("\nString %d is %s", i+1, x[i]);
      }
}
```

Pointers

Session 8

Objectives

- Explain what a pointer is and where it is used
- Explain how to use pointer variables and pointer operators
- Assign values to pointers
- Explain pointer arithmetic
- Explain pointer comparisons
- Explain pointers and single dimensional arrays
- Explain Pointer and multidimensional arrays
- Explain how allocation of memory takes place

What is a Pointer?

- A pointer is a variable, which contains the address of a memory location of another variable
- If one variable contains the address of another variable, the first variable is said to point to the second variable
- A pointer provides an indirect method of accessing the value of a data item
- Pointers can point to variables of other fundamental data types like int, char, or double or data aggregates like arrays or structures

What are Pointers used for?

Some situations where pointers can be used are -

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them instead of moving the arrays itself
- To allocate memory and access it (Direct Memory Allocation)

Pointer Variables

A pointer declaration consists of a base type and a variable name preceded by an *

General declaration syntax is :

```
type *name;
```

For Example:

```
int *var2;
```

Pointer Operators

- There are 2 special operators which are used with pointers :

& and *****

- The & operator is a unary operator and it returns the memory address of the operand

var2 = &var1;

- The second operator * is the complement of &. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value

temp = *var2;

Assigning Values To Pointers-1

- Values can be assigned to pointers through the **&** operator.

```
ptr_var = &var;
```

- Here the address of var is stored in the variable ptr_var
- It is also possible to assign values to pointers through another pointer variable pointing to a data item of the same data type

```
ptr_var = &var;
ptr_var2 = ptr_var;
```

Assigning Values To Pointers-2

- Variables can be assigned values through their pointers as well

***ptr_var = 10;**

- The above declaration will assign 10 to the variable var if ptr_var points to var

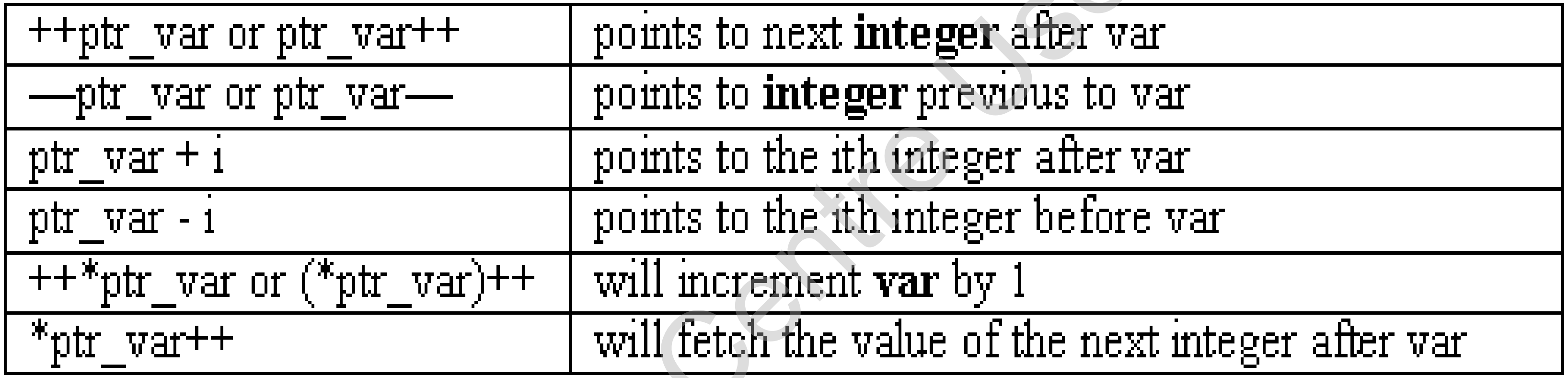
Pointer Arithmetic-1

- Addition and subtraction are the only operations that can be performed on pointers

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
ptr_var++ ;
```

- Let us assume that **var** is stored at the address **1000**
- Then ptr_var has the value 1000 stored in it. Since integers are 2 bytes long, after the expression “ptr_var++;” ptr_var will have the value as 1002 and not 1001

Pointer Arithmetic-2

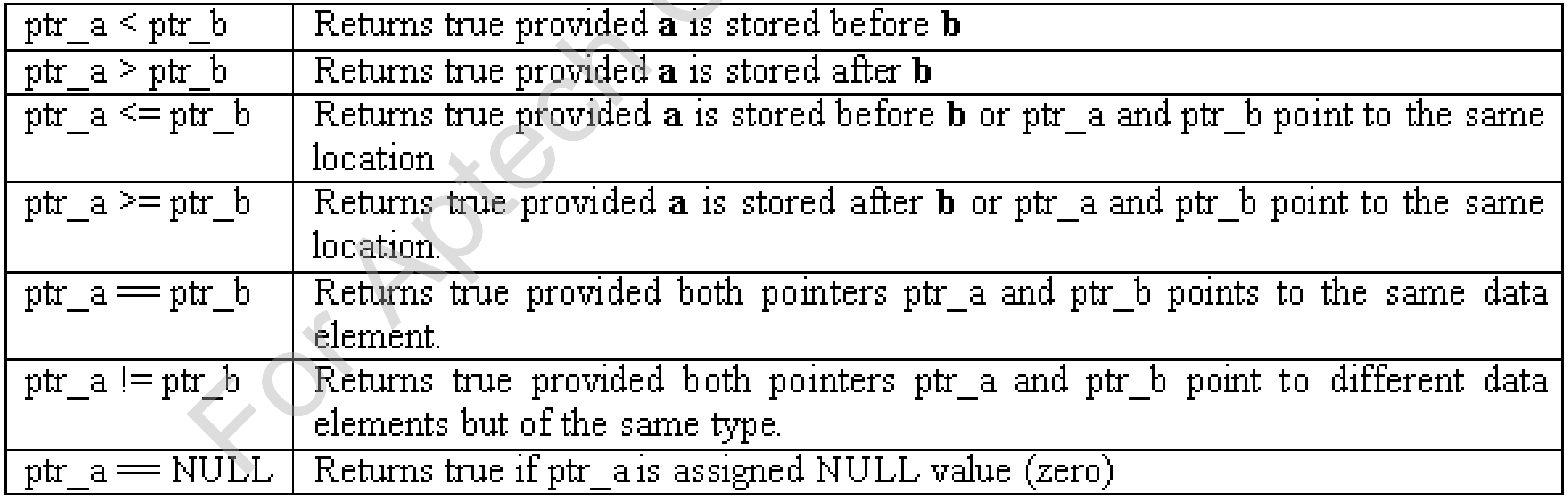


++ptr_var or ptr_var++	points to next **integer** after var
—ptr_var or ptr_var—	points to **integer** previous to var
ptr_var + i	points to the ith integer after var
ptr_var - i	points to the ith integer before var
++*ptr_var or (*ptr_var)++	will increment **var** by 1
*ptr_var++	will fetch the value of the next integer after var

- Each time a pointer is incremented, it points to the memory location of the next element of its base type
- Each time it is decremented it points to the location of the previous element
- All other pointers will increase or decrease depending on the length of the data type they are pointing to

Pointer Comparisons

- Two pointers can be compared in a relational expression provided both the pointers are pointing to variables of the same type
- Consider that ptr_a and ptr_b are 2 pointer variables, which point to data elements a and b. In this case the following comparisons are possible:



ptr_a < ptr_b	Returns true provided **a** is stored before **b**
ptr_a > ptr_b	Returns true provided **a** is stored after **b**
ptr_a <= ptr_b	Returns true provided **a** is stored before **b** or ptr_a and ptr_b point to the same location
ptr_a >= ptr_b	Returns true provided **a** is stored after **b** or ptr_a and ptr_b point to the same location.
ptr_a == ptr_b	Returns true provided both pointers ptr_a and ptr_b points to the same data element.
ptr_a != ptr_b	Returns true provided both pointers ptr_a and ptr_b point to different data elements but of the same type.
ptr_a == NULL	Returns true if ptr_a is assigned NULL value (zero)

Pointers and Single Dimensional Arrays-1

- The address of an array element can be expressed in two ways :
  - By writing the actual array element preceded by the ampersand sign (&)
  - By writing an expression in which the subscript is added to the array name

Pointers and Single Dimensional Arrays-2

Example

```
#include<stdio.h>
void main()
{
       static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
       int i;
       for (i = 0; i < 10; i ++)
       {
           printf(“\ni=%d,aryi]=%d,*(ary+i)=%d“,i,
            ary[i],*(ary + i));
           printf(“&ary[i]= %X,ary+i=%X”,&ary[i],ary+i);
           /* %X gives unsigned hexadecimal  */
       }
}
```

Pointers and Single Dimensional Arrays-3

Output

```
i=0   ary[i]=1   *(ary+i)=1   &ary[i]=194   ary+i = 194
i=1   ary[i]=2   *(ary+i)=2   &ary[i]=196   ary+i = 196
i=2   ary[i]=3   *(ary+i)=3   &ary[i]=198   ary+i = 198
i=3   ary[i]=4   *(ary+i)=4   &ary[i]=19A   ary+i = 19A
i=4   ary[i]=5   *(ary+i)=5   &ary[i]=19C   ary+i = 19C
i=5   ary[i]=6   *(ary+i)=6   &ary[i]=19E   ary+i = 19E
i=6   ary[i]=7   *(ary+i)=7   &ary[i]=1A0   ary+i = 1A0
i=7   ary[i]=8   *(ary+i)=8   &ary[i]=1A2   ary+i = 1A2
i=8   ary[i]=9   *(ary+i)=9   &ary[i]=1A4   ary+i = 1A4
i=9   ary[i]=10  *(ary+i)=10  &ary[i]=1A6   ary+i = 1A6
```

Pointers and Multi Dimensional Arrays-1

- A two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays

- A two-dimensional array declaration can be written as :

```
data_type  (*ptr_var) [expr 2];
```

instead of

```
data_type (*ptr_var) [expr1] [expr 2];
```

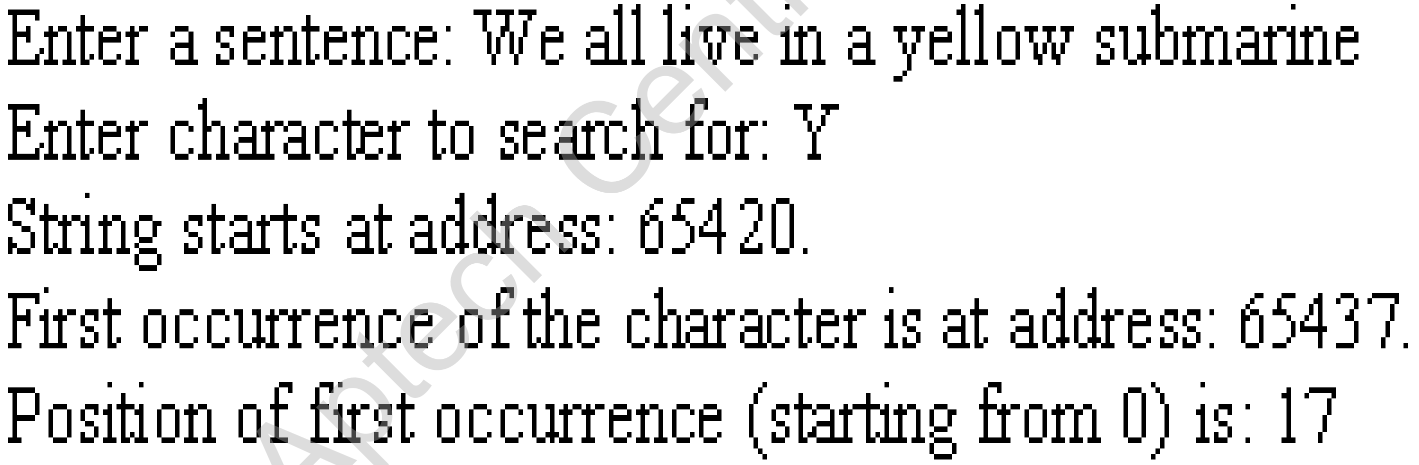
Pointers and Strings-1

Example

```
#include <stdio.h>
#include <string.h>
void main ()
{
        char a, str[81], *ptr;
        printf(“\nEnter a sentence:”);
        gets(str);
        printf(“\nEnter character to search for:”);
        a = getche();
        ptr = strchr(str,a);
        /* return pointer to char*/
        printf( “\nString starts at address: %u”,str);
        printf(“\nFirst occurrence of the character is at
        address: %u ”,ptr);
        printf(“\n Position of first occurrence(starting from
        0)is: % d”, ptr_str);
}
```

Pointers and Strings-2

Output



```
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17
```

Allocating Memory-1

The **malloc()** function is one of the most commonly used functions which permit allocation of memory from the pool of free memory. The parameter for **malloc()** is an integer that specifies the number of bytes needed.

Allocating Memory-2

Example

```
#include<stdio.h>
#include<malloc.h>
void main()
{
int *p,n,i,j,temp;
printf("\n Enter number of elements in the array :");
scanf("%d",&n);
p=(int*)malloc(n*sizeof(int));
for(i=0;i<n;++i) {
 printf("\nEnter element no. %d:",i+1);
 scanf("%d",p+i); }
for(i=0;i<n-1;++i)
  for(j=i+1;j<n;++j)
    if(*(p+i)>*(p+j)) {
         temp=*(p+i);
        *(p+i)=*(p+j);
        *(p+j)=temp;   }
for(i=0;i<n;++i)
        printf("%d\n",*(p+i));
 }
```

free()-1

free() function can be used to de-allocates (frees) memory when it is no longer needed.

Syntax:

```
void free(void *ptr );
```

This function deallocates the space pointed to by *ptr*, freeing it up for future use.

ptr must have been used in a previous call to malloc(), calloc(), or realloc().

free()-2

```
 #include <stdio.h>
 #include <stdlib.h> /*required for the malloc and free functions*/
 int main()
 {
   int number;
   int *ptr;
   int i;
   printf("How many ints would you like store? ");
   scanf("%d", &number);
   ptr = (int *) malloc (number*sizeof(int)); /*allocate memory
*/
 if(ptr!=NULL)
 {
     for(i=0 ; i<number ; i++)
     {
       *(ptr+i) = i;
     }
```

Example

Contd…

free()-3

Example

```
for(i=number ; i>0 ; i--)
    {
       printf("%d\n",*(ptr+(i-1))); /* print out
in reverse order */
    }
    free(ptr); /* free allocated memory */
    return 0;
  }
else
{
    printf("\nMemory allocation failed - not
enough memory.\n");
    return 1;
  }
}
```

calloc()-1

calloc is similar to **malloc**, but the main difference is that the values stored in the allocated memory space is zero by default

- **calloc** requires two arguments
- The first is the number of variables you'd like to allocate memory for
- The second is the size of each variable

Syntax :

```
void *calloc( size_t num, size_t size );
```

calloc()-2

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  float *calloc1, *calloc2;
  int i;
  calloc1 = (float *) calloc(3, sizeof(float));
  calloc2 = (float *)calloc(3, sizeof(float));
  if(calloc1!=NULL && calloc2!=NULL)
  {
    for(i=0 ; i<3 ; i++)
    {
     printf("calloc1[%d] holds %05.5f ", i, calloc1[i]);
      printf("\ncalloc2[%d] holds %05.5f ", i,
            *(calloc2+i));
    }
```

Contd……

calloc()-3

Example

```
   free(calloc1);
   free(calloc2);
   return 0;
}
else
{
     printf("Not enough memory\n");
     return 1;
   }
}
```

realloc()-1

You've allocated a certain number of bytes for an array but later find that you want to add values to it.You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using **realloc**, without losing your data.

- **realloc** takes two arguments
- The first is the pointer referencing the memory
- The second is the total number of bytes you want to reallocate

Syntax:

```
void *realloc( void *ptr, size_t size );
```

realloc()-2

Example

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
 int *ptr;
 int i;
 ptr = (int *)calloc(5, sizeof(int *));
if(ptr!=NULL)
{
  *ptr = 1;  *(ptr+1) = 2;
  ptr[2] = 4;  ptr[3] = 8;  ptr[4] = 16;
  ptr = (int *)realloc(ptr, 7*sizeof(int));
  if(ptr!=NULL)
  {
    printf("Now allocating more memory... \n");
   ptr[5] = 32; /* now it's legal! */
   ptr[6] = 64;
```

realloc()-3

Example

```
for(i=0 ; i<7 ; i++)
      {
        printf("ptr[%d] holds %d\n", i, ptr[i]);
      }
      realloc(ptr,0); /* same as free(ptr); - just fancier! */
      return 0;
    }
    else
    {
      printf("Not enough memory - realloc failed.\n");
      return 1;
    }
  }
else
{
    printf("Not enough memory - calloc failed.\n");
    return 1;
  }
}
```

Functions

Session 9

Objectives

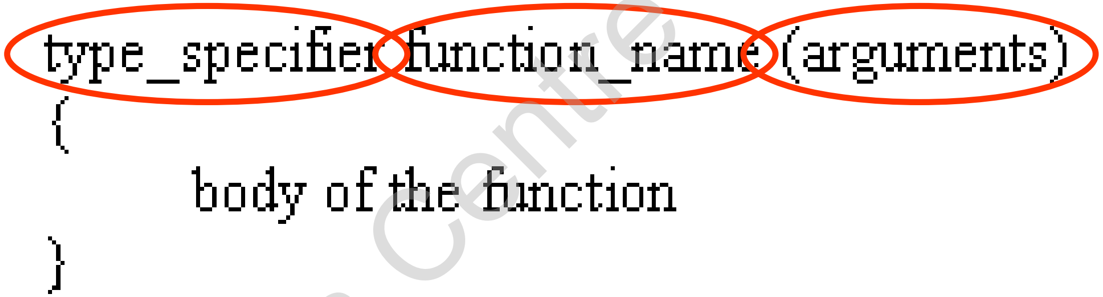
- Explain the use of functions
- Explain the structure of a function
- Explain function declaration and function prototypes
- Explain the different types of variables
- Explain how to call functions
- Call by Value
- Call by Reference
- Explain the scope rules for a function
- Explain functions in multifile programs
- Explain Storage classes
- Explain function pointers

Functions

- A function is a self-contained program segment that carries out a specific, well-defined task
- Functions are generally used as abbreviations for a series of instructions that are to be executed more than once
- Functions are easy to write and understand
- Debugging the program becomes easier as the structure of the program is more apparent, due to its modular form
- Programs containing functions are also easier to maintain, because modifications, if required, are confined to certain functions within the program

The Function Structure

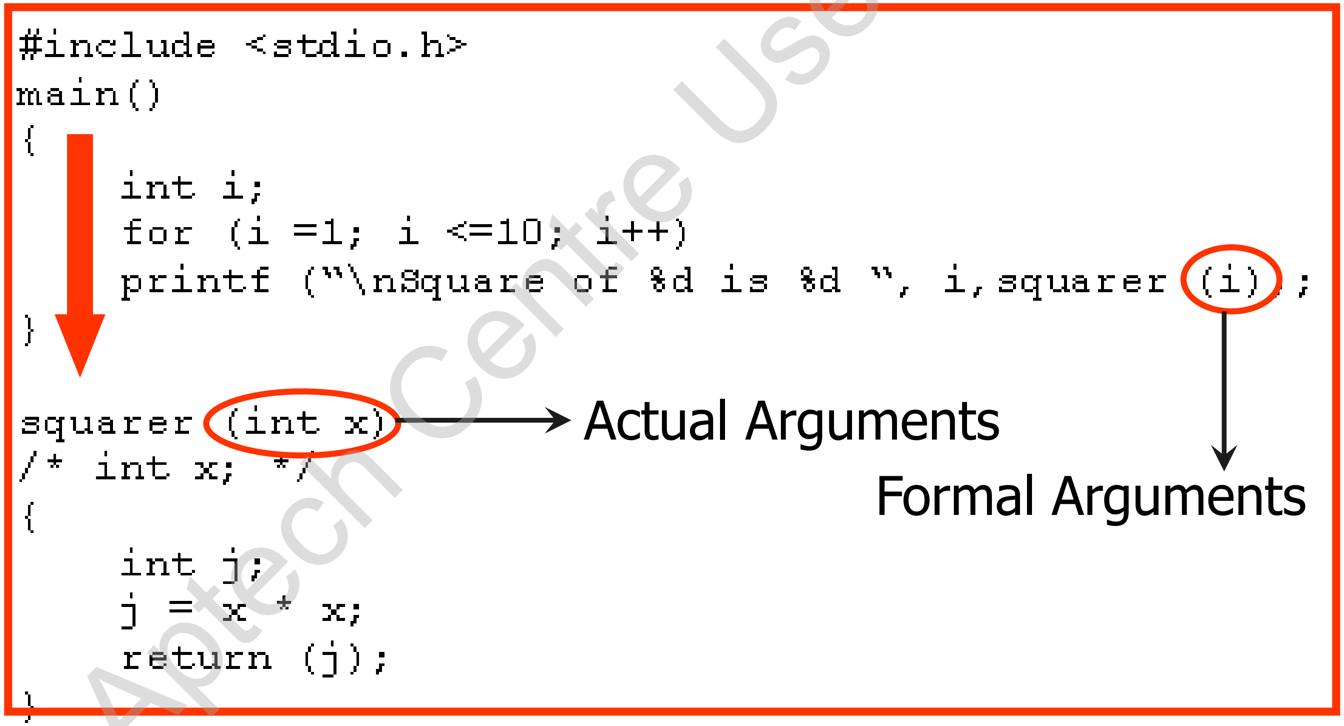
- The general syntax of a function in C is :



```
type_specifier function_name (arguments)
{
        body of the function
}
```

- The type_specifier specifies the data type of the value, which the function will return.
- A valid function name is to be assigned to identify the function
- Arguments appearing in parentheses are also termed as formal parameters.

Arguments of a function



```
#include <stdio.h>
main()
{
    int i;
    for (i =1; i <=10; i++)
    printf (“\nSquare of %d is %d “, i,squarer (i));
}

squarer (int x)
/* int x; */
{
    int j;
    j = x * x;
    return (j);
}
```

- The program calculates the square of numbers from 1 to 10
- The data is passed from the main() to the squarer() function
- The function works on data using arguments

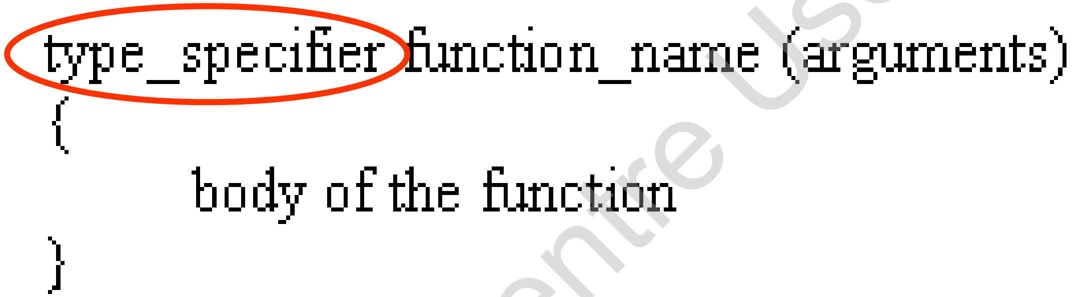
Returning from the function

```
squarer (int x)
/* int x; */
{
    int j;
    j = x * x;
    return (j);
}
```

- It transfers the control from the function back to the calling program immediately.

- Whatever is inside the parentheses following the return statement is returned as a value to the calling program.

Data Type of a Function



```
type_specifier function_name (arguments)
{
        body of the function
}
```

- The type_specifier is not written prior to the function squarer(), because squarer() returns an integer type value
- The type_specifier is not compulsory if an integer type of value is returned or if no value is returned
- However, to avoid inconsistencies, a data type should be specified

Invoking a Function

- A semicolon is used at the end of the statement when a function is called, but not after the function definition
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not
- Only one value can be returned by a function
- The program can have more than one function
- The function that calls another function is known as the calling function/routine
- The function being called is known as the called function/routine

Function Declaration

- Declaring a function becomes compulsory when the function is being used before its definition

- The address() function is called before it is defined

- Some C compilers return an error, if the function is not declared before calling

- This is sometimes referred to as Implicit declaration

```
#include <stdio.h>
main()
{
        .
        .
        address()
        .
        .
}

address()
{
        .
        .
        .
}
```

Function Prototypes

- Specifies the data types of the arguments

char abc(int x, nt y);

Advantage :

Any illegal type conversions between the arguments used to call a function and the type definition of its parameters is reported

char noparam (void);

Variables

- Local Variables
  - Declared inside a function
  - Created upon entry into a block and destroyed upon exit from the block
- Formal Parameters
  - Declared in the definition of function as parameters
  - Act like any local variable inside a function
- Global Variables
  - Declared outside all functions
  - Holds value throughout the execution of the program

Storage Classes-1

- Every C variable has a characteristic called as a storage class
- The storage class defines two characteristics of the variable:
  - **Lifetime** – The lifetime of a variable is the length of time it retains a particular value
  - **Visibility** – The visibility of a variable defines the parts of a program that will be able to recognize the variable

Storage Classes-2

- **auto**matic
- **exter**nal
- **static**
- **register**

Function Scope rules

- Scope Rules - Rules that govern whether one piece of code knows about or has access to another piece of code or data
- The code within a function is private or local to that function
- Two functions have different scopes
- Two Functions are at the same scope level
- One function cannot be defined within another function

Calling The Functions

- Call by value
- Call by reference

Calling By Value

- In C, by default, all function arguments are passed by value

- When arguments are passed to the called function, the values are passed through temporary variables

- All manipulations are done on these temporary variables only

- The arguments are said to be passed by value when the value of the variable are passed to the called function and any alteration on this value has no effect on the original value of the passed variable

Calling By Reference

- In call by reference, the function is allowed access to the actual memory location of the argument and therefore can change the value of the arguments of the calling routine

- Definition

```
getstr(char *ptr_str, int *ptr_int);
```

- Call

```
getstr(pstr, &var);
```

Nesting Function Calls

```
main()
  {
    .
    .

  palindrome();
    .
    .
  }
```

```
palindrome()
  {
      .
      .
      getstr();
      reverse();
      cmp();
      .
      .
  }
```

Functions in Multifile Programs

- Functions can also be defined as **static** or **external**
- Static functions are recognized only within the program file and their scope does not extend outside the program file

```
static fn _type fn_name (argument list);
```

- External function are recognized through all the files of the program

```
extern fn_type fn_name (argument list);
```

Function Pointers

- Address is the entry point of the function
- Function has a physical location in memory that can be assigned to a pointer

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b, int (*cmp)());
main()
{
		char sl[80];
		int (*p)();
		p = strcmp;
		gets(s1);
		gets(s2);
		check(s1, s2, p);
}
```

```
void check(char *a, char *b, int (*cmp)())
{
printf(“testing for equality \n”);
if (!(*cmp)(a,b))
		printf(“Equal”);
else
		printf(“Not Equal”);
}
```

Strings

Session 10

Objectives

- Explain string variables and constants
- Explain pointers to strings
- Perform string input/output operations
- Explain the various string functions
- Explain how arrays can be passed as arguments to functions
- Describe how strings can be used as function arguments

String variables

- **Strings are arrays of characters terminated by the NULL ('\0') character.**
- **String variables can be assigned string constants.**
- **A string constant is a sequence of characters surrounded by double quotes.**
- **The '\0' null character is automatically added in the internal representation of a string.**
- **While declaring a string variable, allow one extra element space for the null terminator.**

Declaring string variables

- A typical string variable declaration is:.

```
char str[10];
```

- **str** is a character array variable that can hold a maximum of 10 characters including the null terminator.

String I/O operations-1

- String I/O operations are carried out using functions from the standard I/O library called **stdio.h**
- The gets() function is the simplest method of accepting a string through standard input
- Input characters are accepted till the Enter key is pressed
- The gets() function replaces the terminating '\n' new line character with the '\0' character
- Syntax :

gets(str);

String I/O operations-2

- The puts() function is used to display a string on the standard output device.
- Syntax :

puts(str);

- The scanf() and printf() functions are used to accept and display mixed data types with a single statement.
- The syntax to accept a string is as follows:

scanf(“%s”, str);

- The syntax to display a string is as follows:

printf(“%s”, str);

String Functions

Functions for handling strings are found in the standard header file **string.h**. Few of the operations performed by these functions are:

- Concatenating strings
- Comparing strings
- Locating a character in a string
- Copying one string to another
- Calculating the length of a string

The strcat() function

- Joins two string values into one.
- Syntax:

strcat(str1, str2);

- Concatenates the str2 at the end of str1
- The function returns str1

The strcmp() function

- Compares two strings and returns an integer value based on the results of the comparison.

- Syntax:

  strcmp(str1, str2);

- The function returns a value:
  - Less than zero if str1<str2
  - Zero if str1 is same as str2
  - Greater than zero if str1>str2

The strchr() function

- Determines the occurrence of a character in a string.
- Syntax:

  `strchr(str, chr);`

- The function returns a value:
  - Pointer to the first occurrence of the character (pointed by **chr**) in the string, **str**
  - NULL if it is not present

The strcpy() function

- Copies the value in one string onto another

- Syntax:

```
strcpy(str1, str2);
```

- The value of str2 is copied onto str1

- The function returns **str1**

The strlen() function

- Determines the length of a string

- Syntax:

  strlen(str);

- The function returns an integer value for the length of **str**

Passing Arrays to Functions-1

- When an array is passed as an argument to a function, only the address of the array is passed

- The array name without the subscripts refers to the address of the array

```
void main()
      {
    int ary[10];
          .
          .
    fn_ary(ary);
          .
          .
      }
```

Passing Arrays to Functions-2

```
#include<stdio.h>

void  main()
{
int num[5], ctr, sum=0;
int sum_arr(int num_arr[]); /* Function declaration */

   clrscr();

   for(ctr=0;ctr<5;ctr++) /* Accepts numbers into the array */
   {
            printf("\nEnter number %d: ", ctr+1);
            scanf("%d", &num[ctr]);
   }
```

Passing Arrays to Functions-3

```
   sum=sum_arr(num); /* Invokes the function */

   printf("\nThe sum of the array is %d", sum);

   getch();
}

int sum_arr(int num_arr[]) /* Function definition */
{
   int i, total;

   for(i=0,total=0;i<5;i++) /* Calculates the sum */
          total+=num_arr[i];

   return total; /* Returns the sum to main() */
}
```

Passing Arrays to Functions-4

Sample output of the program

```
Enter number 1: 5

Enter number 2: 10

Enter number 3: 13

Enter number 4: 26

Enter number 5: 21

The sum of the array is 75
```

Example of Passing Strings to Functions-1

```
#include<stdio.h>
#include<string.h>

void  main()
{
char lines[5][20];
int ctr, longctr=0;
int longest(char lines_arr[][20]);
/* Function declaration */

   clrscr();
   for(ctr=0;ctr<5;ctr++)
   /* Accepts string values into the array */
   {
            printf("\nEnter string %d: ", ctr+1);
            scanf("%s", lines[ctr]);
   }
```

Example of Passing Strings to Functions-2

```
   longctr=longest(lines);
   /* Passes the array to the function */

   printf("\nThe longest string is %s", lines[longctr]);

   getch();
}

int longest(char lines_arr[][20]) /* Function definition */
{
   int i=0, l_ctr=0, prev_len, new_len;

   prev_len=strlen(lines_arr[i]);
   /* Determines the length of the first element */
```

Example of Passing Strings to Functions-3

```
  for(i++;i<5;i++)
  {
          new_len=strlen(lines_arr[i]);
          /* Determines the length of the next element */

          if(new_len>prev_len)
                    l_ctr=i;
/* Stores the subscript of the longer string */

          prev_len=new_len;
  }

  return l_ctr;
  /* Returns the subscript of the longest string */
}
```

Example of Passing Strings to Functions-4

Sample output of the program

```
Enter string 1: The

Enter string 2: Sigma

Enter string 3: Protocol

Enter string 4: Robert

Enter string 5: Ludlum

The longest string is Protocol
```

Advanced Data types and Sorting

Session 11

Objectives - 1

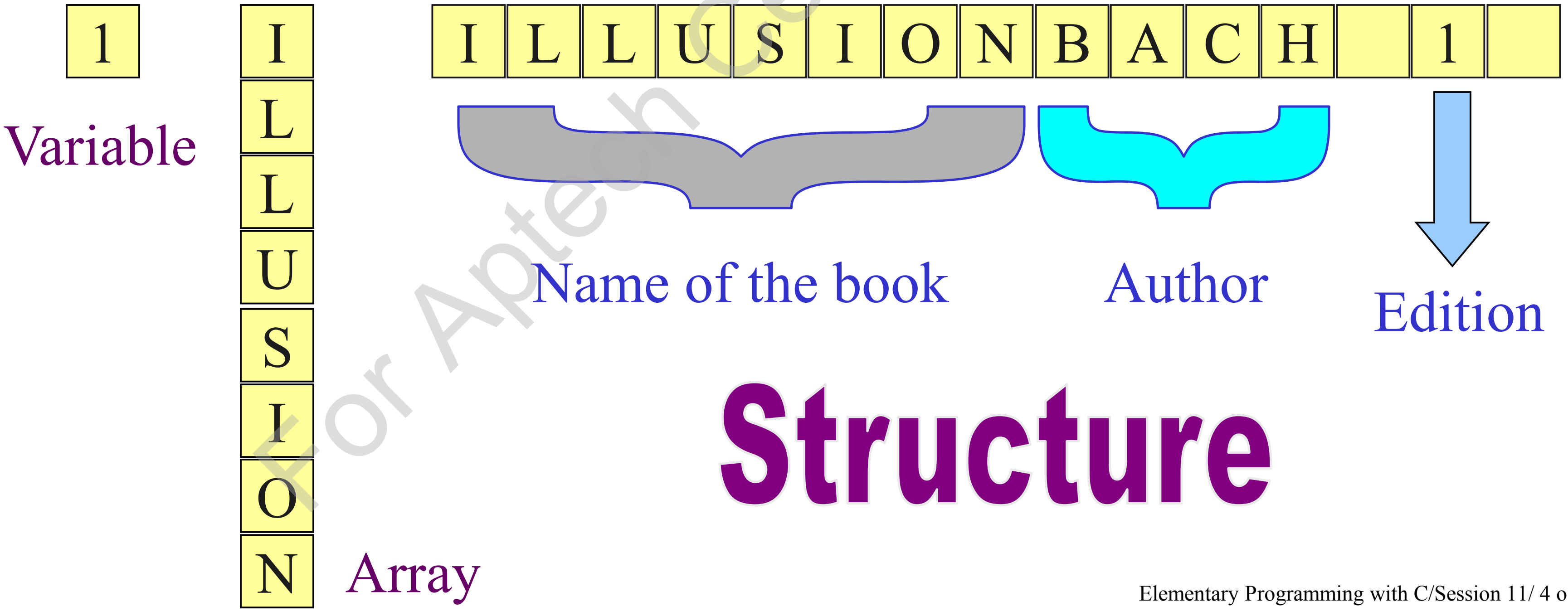
- Explain structures and their use
- Define structures
- Declare structure variables
- Explain how structure elements are accessed
- Explain how structures are initialized
- Explain how assignment statements are used with structures
- Explain how structures can be passed as arguments to functions
- Use arrays of structures
- Explain the initialization of structure arrays

Objectives - 2

- Explain pointers to structures
  Explain how structure pointers can be passed as arguments to functions
- Explain the typedef keyword
- Explain array sorting with the Selection sort and Bubble sort methods

Structures

- A structure consists of a number of data items, which need not be of the same data type, grouped together
- The structure could hold as many of these items as desired



Defining a Structure

- A structure definition forms a template for creating structure variables
- The variables in the structure are called structure elements or structure members
- Example:

```
struct cat
    {     char bk_name [25];
          char author [20];
          int edn;
          float price;
     };
```

Declaring Structure Variables

- Once the structure has been defined, one or more variables of that type can be declared
- Example: **`struct cat books1;`**
- The statement sets aside enough memory to hold all items in the structure

Other ways

```
struct cat {    char bk_name[25];
                char author[20];
                int edn;
                float price;
        } books1, books2;
```

```
struct cat books1, books2;
```

or

```
struct cat books1;
struct cat books2;
```

Accessing Structure Elements

- Structure elements are referenced through the use of the **dot operator** (.), also known as the **membership operator**
- Syntax:

```
structure_name.element_name
```

- Example:

```
scanf(“%s”, books1.bk_name);
```

Initializing Structures

- Like variables and arrays, structure variables can be initialized at the point of declaration

```
 struct employee
   {         int no;
     char name [20];
   };
```

- Variables **emp1** and **emp2** of the type **employee** can be declared and initialized as:

```
struct employee emp1 = {346, “Abraham”};
struct employee emp2 = {347, “John”};
```

Assignment Statements Used with Structures-1

- It is possible to assign the values of one structure variable to another variable of the same type using a simple assignment statement

- For example, if **books 1** and **books2** are structure variables of the same type, the following statement is valid

```
books2 = books1;
```

Assignment Statements Used with Structures - 2

- In cases where direct assignment is not possible, the in-built function **memcpy()** can be used

- Syntax:

memcpy (char * destn, char &source, int nbytes);

- Example:

memcpy (&books2, &books1, sizeof(struct cat));

Structures within Structures

- **It is possible to have one structure within another structure. A structure cannot be nested within itself**

```
struct issue
{
   char borrower [20];
   char dt_of_issue[8];
   struct cat books;
}issl;
```

- **To access the elements of the structure the format will be similar to the one used with normal structures,**

```
issl.borrower
```

- **To access elements of the structure cat, which is a part of another structure issue,**

```
issl.books.author
```

Passing Structures as Arguments

- A structure variable can be passed as an argument to a function

- This facility is used to pass groups of logically related data items together instead of passing them one by one

- The type of the argument should match the type of the parameter

Array of Structures

- A common use of structures is in arrays of structures
- A structure is first defined, and then an array variable of that type is declared
- Example:

```
struct cat books[50];
```

- To the access the variable author of the fourth element of the array **books**:

```
books[4].author
```

Initialization of Structure Arrays

- Structure arrays are initialized by enclosing the list of values of its elements within a pair of braces
- Example:

```
struct unit
{     char ch;
      int i;
      };

     struct unit series [3] =
{     {‘a’, 100}
      {‘b’, 200}
      {‘c’, 300}
     };
```

Pointers to Structures

- Structure pointers are declared by placing an asterisk(*) in front of the structure variable’s name
- The **->** operator is used to access the elements of a structure using a pointer
- Example:

```
struct cat *ptr_bk;
ptr_bk = &books;
printf(“%s”, ptr_bk->author);
```

- Structure pointers passed as arguments to functions enable the function to modify the structure elements directly

The **typedef** keyword

- A new data type name can be defined by using the keyword typedef
- It does not create a new data type, but defines a new name for an existing type
- Syntax:

```
typedef type name;
```

- Example:

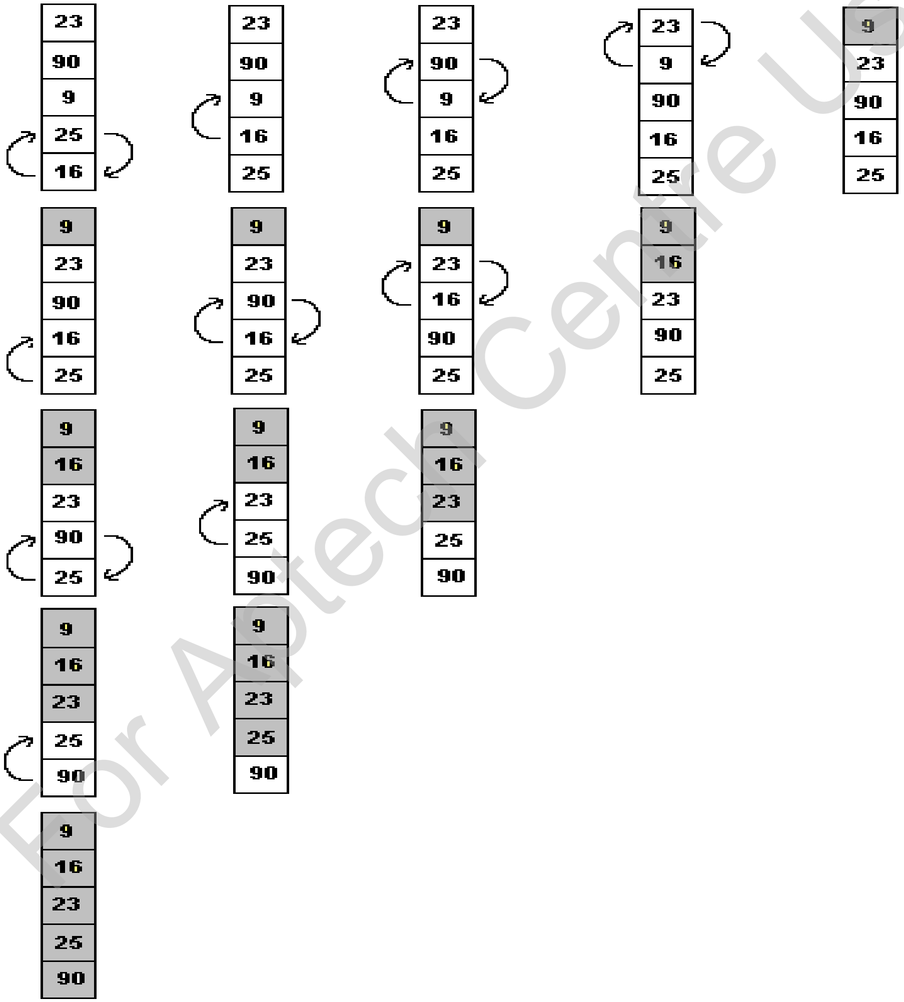
```
typedef float deci;
```

- typedef cannot be used with storage classes

Sorting Arrays

- Sorting involves arranging the array data in a specified order such as ascending or descending
- Data in an array is easier to search when the array is sorted
- There are two methods to sort arrays – Selection Sort and Bubble Sort
- In the selection sort method, the value present in each element is compared with the subsequent elements in the array to obtain the least/greatest value
- In bubble sort method, the comparisons begin from the bottom-most element and the smaller element bubbles up to the top

Bubble Sort-1



Bubble Sort-2

Example

```
#include <stdio.h>

void main()
{
  int i, j, temp, arr_num[5] = { 23, 90, 9, 25, 16};

  clrscr();
  for(i=3;i>=0;i--) /* Tracks every pass */
       for(j=4;j>=4-i;j--) /* Compares elements */
              {     if(arr_num[j]<arr_num[j-1])
                    {      temp=arr_num[j];
                           arr_num[j]=arr_num[j-1];
                    arr_num[j-1]=temp;
                    }
              }
```

Contd.....

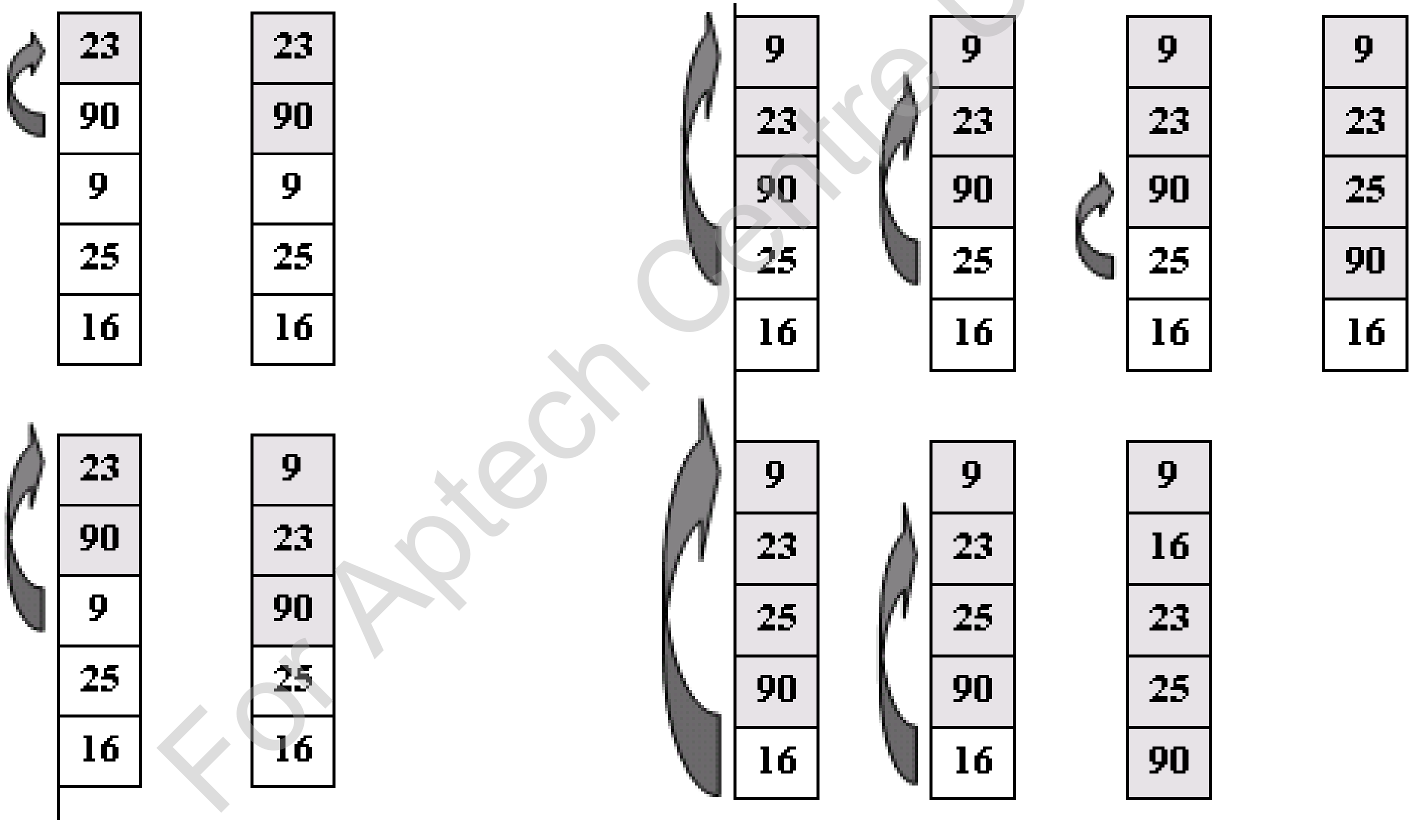
Bubble Sort-3

```
    printf("\nThe sorted array");
    for(i=0;i<5;i++)
                printf("\n%d", arr_num[i]);

    getch();
}
```

Example

Insertion Sort-1



Insertion Sort-2

```
#include<stdio.h>
void main()
{
		int i, j, arr[5] = { 23, 90, 9, 25, 16 };
		char flag;
		clrscr();
		/*Loop to compare each element of the unsorted part of the array*/
		for(i=1; i<5; i++)
				/*Loop for each element in the sorted part of the array*/
				for(j=0, flag='n'; j<i && flag=='n'; j++)
				{
						if(arr[j]>arr[i])
						{	/*Invoke the function to insert the number*/
								insertnum(arr, i, j);
								flag='y';
						}
				}
		printf("\n\nThe sorted array\n");
		for(i=0; i<5; i++)
				printf("%d\t", arr[i]);
		getch();
}
```

Insertion Sort-3

```
insertnum(int arrnum[], int x, int y)
{
        int temp;
        /*Store the number to be inserted*/
        temp=arrnum[x];
        /*Loop to push the sorted part of the array down from the position
where the number has to inserted*/
        for(;x>y; x--)
                arrnum[x]=arrnum[x-1];
        /*Insert the number*/
        arrnum[x]=temp;
}
```

File Handling

Session 12

Objectives

- Explain streams and files
- Discuss text streams and binary streams
- Explain the various file functions
- Explain file pointer
- Discuss current active pointer
- Explain command-line arguments

File Input/Output

- All I/O operations in C are carried out using functions from the standard library

- This approach makes the C file system very powerful and flexible

- I/O in C is unique because data may be transferred in its internal binary representation or in a human-readable text format

Streams

- The C file system works with a wide variety of devices including printers, disk drives, tape drives and terminals

- Though all these devices are very different from each other, the buffered file system transforms each device into a logical device called a stream

- Since all streams act similarly, it is easy to handle the different devices

- There are two types of streams - the text and binary streams

Text Streams

- A text stream is a sequence of characters that can be organized into lines terminated by a new line character

- In a text stream, certain character translations may occur as required by the environment

- Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those in the external device

- Also, because of possible translations, the number of characters written (or read) may not be the same as those in the external device

Binary Streams

- A binary stream is a sequence of bytes with a one-to-one correspondence to those in the external device, that is, there are no character translations

- The number of bytes written (or read) is the same as the number on the external device

- Binary streams are a flat sequence of bytes, which do not have any flags to indicate the end of file or end of record

- The end of file is determined by the size of the file

Files

- A file can refer to anything from a disk file to a terminal or a printer
- A file is associated with a stream by performing an open operation and disassociated by a close operation
- When a program terminates normally, all files are automatically closed
- When a program crashes, the files remain open

Basic File Functions

Name	Function
fopen( )	Opens a file
fclose( )	Closes a file
fputc( )	Writes a character to a file
fgetc( )	Reads a character from a file
fread()	Reads from a file to a buffer
fwrite()	Writes from a buffer to a file
fseek( )	Seeks a specific location in the file
fprintf( )	Operates like printf(), but on a file
fscanf( )	Operates like scanf(), but on a file
feof( )	Returns true if end-of-file is reached
ferror( )	Returns true if an error has occurred
rewind( )	Resets the file position locator to the beginning of the file
remove( )	Erases a file
fflush( )	Writes data from internal buffers to a specified file

File Pointer

- A file pointer is essential for reading or writing files

- It is a pointer to a structure that contains the file name, current position of the file, whether the file is being read or written, and whether any errors or the end of the file have occurred

- The definitions obtained from stdio.h include a structure declaration called FILE

- The only declaration needed for a file pointer is:

FILE *fp

Opening a Text File

- The fopen() function opens a stream for use and links a file with that stream
- The fopen() function returns a file pointer associated with the file
- The prototype for the fopen() function is:
  FILE *fopen(const char *filename, const char *mode);

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+f	Append or create a text file for read/write

Closing a Text File

- It is important to close a file once it has been used
- This frees system resources and reduces the risk of overshooting the limit of files that can be open
- Closing a stream flushes out any associated buffer, an important operation that prevents loss of data when writing to a disk
- The fclose() function closes a stream that was opened by a call to fopen()
- The prototype for fclose() is:

```
int  fclose(FILE *fp);
```

- The fcloseall() function closes all open streams

Writing a Character – Text File

- Streams can be written to either character by character or as strings

- The fputc() function is used for writing characters to a file previously opened by fopen()

- The prototype is:

```
int fputc(int ch, FILE *fp);
```

Reading a Character – Text File

- The fgetc() function is used for reading characters from a file opened in read mode, using fopen()
- The prototype is:

  int fgetc(int ch, FILE *fp);

- The fgetc() function returns the next character from the current position in the input stream, and increments the file position indicator

String I/O

- The functions fputs() and fgets() write and read character strings to and from a disk file
- The fputs() function writes the entire string to the specified stream
- The fgets() function reads a string from the specified stream until either a new line character is read or length-1 characters have been read
- The prototypes are:

int fputs(const char *str, FILE *fp);

char *fgets(char *str, int length, FILE *fp);

Opening a File-Binary

- The fopen() function opens a stream for use and links a file with that stream
- The fopen() function returns a file pointer associated with the file
- The prototype for the fopen() function is:
  `FILE *fopen(const char *filename, const char  *mode);`

Mode	Meaning
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append a binary file for read/write

Closing a File Binary

- The fclose() function closes a stream that was opened by a call to fopen()

- The prototype for fclose() is:

int fclose(FILE *fp);

The fread() and fwrite() functions

- The functions fread() and fwrite() are referred to as unformatted read or write functions
- They are used to read and write an entire block of data to and from a file
- The most useful application involves reading and writing user-defined data types, especially structures
- The prototypes for the functions are:

```
size_t fread(void *buffer, size_t num_bytes, size_t count,
FILE *fp);
```

```
size_t fwrite(const void *buffer, size_t num_bytes, size_t
count, FILE *fp);
```

Using feof()

- The function feof() returns true if the end of the file has been reached, otherwise it returns false (0)

- This function is used while reading binary data

- The prototype is:

  int feof (FILE *fp);

The rewind() function

- The rewind() function resets the file position indicator to the beginning of the file
- It takes the file pointer as its argument
- Syntax:

```
rewind(fp );
```

The ferror() function

- The ferror() function determines whether a file operation has produced an error

- As each operation sets the error condition, ferror() should be called immediately after each operation; otherwise, an error may be lost

- Its prototype is:

  int ferror(FILE *fp);

Erasing Files

- The remove() function erases a specified file
- Its prototype is:

```
int remove(char *filename);
```

Flushing streams

- The fflush() function flushes out the buffer depending upon the file type
- A file opened for read will have its input buffer cleared, while a file opened for write will have its output buffer written to the files
- Its prototype is:

  int fflush(FILE *fp);

- The fflush() function, with a null, flushes all files opened for output

The Standard Streams

Whenever a C program starts execution under DOS, five special streams are opened automatically by the operating system

- **The standard input (stdin)**
- **The standard output (stdout)**
- **The standard error (stderr)**
- **The standard printer (stdprn)**
- **The standard auxiliary (stdaux)**

Current Active Pointer

- A pointer is maintained in the FILE structure to keep track of the position where I/O operations take place
- Whenever a character is read from or written to the stream, the current active pointer (known as curp) is advanced
- The current location of the current active pointer can be found with the help of the ftell() function
- The prototype is:

long int ftell(FILE *fp);

Setting Current Position-1

- The fseek() function repositions the curp by the specified number bytes from the start, the current position or the end of the stream depending upon the position specified in the fseek() function

- The prototype is:

int fseek (FILE *fp, long int offset, int origin);

Setting Current Position-2

- The origin indicates the starting position of the search and has values as follows:

Origin	File Location
SEEK_SET or 0	Beginning of file
SEEK_CUR or 1	Current file pointer position
SEEK_END or 2	End of file

fprintf() and fscanf()-1

- The buffered I/O system includes fprintf() and fscanf() functions that are similar to printf() and scanf() except that they operate with files

- The prototypes of are:

```
int fprintf(FILE * fp, const char *control_string,..);

int fscanf(FILE *fp, const char *control_string,...);
```

fprintf() and fscanf()-2

- The fprintf() and fscanf() though the easiest, are not always the most efficient

- Extra overhead is incurred with each call, since the data is written in formatted ASCII data instead of binary format

- So, if speed or file size is a concern, fread() and fwrite() are a better choice